STRUCTURED DOCUMENTS

The Cambridge Series on Electronic Publishing

Editor: *P. Hammersley*

The purpose of this new series is to publish books in the exciting and topical field of electronic printing and publishing. The series will attempt to cover all aspects of electronic publishing, including:

- Matters relating to hardware, software and standards concerned with transferring authors' words and images to the printed page;
- Document delivery – electronic methods of reader access to publications, full-text database publishing, CD-ROM delivery systems and the impact for libraries;
- On-line publishing systems and distributed publication systems—the impact of networks on the publication process;
- Human factors, relating, for example, to document structures for readability, editing aids, expert system-based interfaces and type and page design.

While there will be some overlap with other areas of computer science (such as information retrieval and computer graphics), the series will concentrate primarily on works concerned with publishing, and so should provide a scholarly and specific survey of developments in this subject-area.

Titles will range from graduate-level texts to multi-author edited works.

Already published:
Document Manipulation and Typography; J. C. van Vliet (ed)

STRUCTURED DOCUMENTS

Edited by

J. ANDRÉ

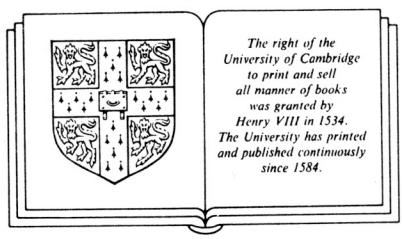
INRIA, Rennes

R. FURUTA

University of Maryland

V. QUINT

INRIA, Grenoble



CAMBRIDGE UNIVERSITY PRESS

Cambridge

New York New Rochelle

Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York NY 10022, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

© Cambridge University Press 1989

First published 1989

Printed in Great Britain by Billing & Sons Ltd, Worcester

British Library cataloguing in publication data

Structured documents. – (Cambridge series on electronic publishing; 2).
1. Computer systems. Data. Representation
I. André, Jacques II. Furuta, Richard
III. Quint, Vincent
005.7'2

Library of Congress cataloguing in publication data available

ISBN 0 521 36554 6

Preface

A document may be described as a collection of objects with higher-level objects formed from more primitive objects. The object relationships represent the logical relationships between components of the document. For example, the present document is described as a book at the highest level. The book is subdivided into chapters, each chapter into sections, subsections, paragraphs, and so forth. Such a document organization has come to be known as the *structured document* representation.

This book describes the principles underlying this logical structuring of documents and the advantages that can be drawn from it.

The book originated with a series of lectures given during a winter school on document structures organized by INRIA.[1] The school, directed by Jacques André and Vincent Quint, took place at Aussois (Savoy, France) in January 1987. The course brought together sixty or so researchers and engineers involved in the development or use of computer systems handling documents and their structures. Most of the lectures presented at this school were accompanied by a draft document, collected together in a booklet distributed to the school's participants [AQ87].[2] These draft documents have formed the basis for the present book.[3]

[1] The *Institut National de Recherche en Informatique et Automatique* is the French national laboratory for research in Computer Science. For more information on INRIA and its schools, write to *INRIA, Service des relations extérieures, B.P. 105, F-78153 Le Chesnay Cedex, France.*

[2] Bibliographic citations, shown between square brackets, found in this chapter and through the book refer to the collected bibliography located at the end of the book.

[3] Several papers were given during the course which could not, for a variety of reasons, be included in this book. In particular, these included a paper by Philippe Maurice from the Caen SEPT on ODA, a paper by Jacques André on document systems compared to programming

The book reflects the structure and contents of the course: four survey papers giving a fairly complete overview of the field, accompanied by shorter papers, each of which gives a particular viewpoint on the problem. The contributions as a whole, and therefore the corresponding chapters of this book, were specifically designed for the course. We will describe the organization of the book in more detail in the first chapter.

Our heartfelt thanks are due to David Beeson, ably assisted by Janet Macke, who translated those texts that were originally produced in French.[4] We are heavily indebted to the INRIA for financing this project, and to Agnès Couaillier and Valéry Brucker for their help in the Aussois school organization. Paul Gatewood prepared software at the University of Maryland that assisted in creating the book's merged bibliography and Philippe Louarn of IRISA helped us produce the camera-ready output in Rennes. We thank members of the staff in the Systems and Software Technology Division of the (United States) National Bureau of Standards for their time in discussion of relevant issues. And of course, we are indebted to the participants in the Document Structure course, whose efforts gave the course its value and to the lecturers who gave this book its contents.

Jacques André
Richard Furuta
Vincent Quint

languages, three notes respectively from Jacques André, Pat Norrish and Richard Southall on tabular typography, and finally one by Isabelle Levy (Italsoft, Paris) on *Text and logical markup for professional publishers.*

[4]The course was bilingual—presentations and papers were presented in English or French at the preference of the lecturer.

Table of Contents

By Way of an Introduction. Structured Documents: What and why? 1
Jacques André, Richard Furuta, and Vincent Quint

Concepts and Models for Structured Documents 7
Richard Furuta

Systems for the Manipulation of Structured Documents 39
Vincent Quint

Document Representation: Concepts and standards 75
Vania Joloboff

Electronic Mail of Structured Documents:
Representation, transmission, and archiving 107
Brian K. Reid

Interfaces Between the Designer and the Document 119
Richard Southall

Text Structure Recognition in Optical Reading 133
Rolf Ingold

Semantic Structures of Text 143
Pat Norrish

The Contribution of Linguistic Knowledge to the Interpretation of Text Structures 161
Jacques Virbel

Performing Textual Structures in Documents 181
Saïd Tazi

By Way of a Colophon 191

Bibliography 193

Index 215

By Way of an Introduction. Structured Documents: What and why?

Jacques André*
Richard Furuta†
Vincent Quint‡

Later in this chapter, we will describe the organization of the book in more detail and introduce the individual chapters that follow. First, in order to give a clearer definition of the subject, we briefly will summarize the developments that led to the concepts and tools with which we are concerned here.

1 A little history

Document production systems started timidly in the 1960s. The first systems were simple extensions to program editing tools, for example formatters such as ROFF or RUNOFF. They then grew rapidly and now form a class of very varied tools.

The first systems were low-level formatters, handling text input through the editor used for writing programs. The role of these formatters was very simple. Essentially it was to construct lines of equal length and to produce justified pages on the basis of a *ribbon* of text. A few commands could be introduced into the text to control page layout: these were elementary commands, very close to those understood by the printer (spaces, linefeeds, formfeeds, etc.). The facilities provided by printers of the period necessarily limited the potential for formatting: a limited character set, fixed pitch and only rough positioning of characters.

The language in which commands were expressed showed little formalism and was at a low level, similar to an assembly language. These lan-

*INRIA/IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France
†Department of Computer Science, University of Maryland, College Park, MD 20742, U.S.A.
‡INRIA/IMAG-LGI, BP 53X F-38041 Grenoble Cedex, France

guages have advanced in line with developments in programming languages. To start with, macro-instructions were added, making it possible to adapt the formatter and to provide the user with higher-level commands. The commands too were enhanced to take greater advantage of the possibilities provided by new printers, or even phototypesetters, as in TROFF [KLO78].

Formatters were also enhanced to handle non-text elements. The formatting tools available under UNIX [Rit78] provide a good example of this development. A set of preprocessors handles various types of objects within documents intended for TROFF: Tbl [Les76] handles tables, Eqn [KC75] handles formulas, Pic [Ker82] handles drawings. Other tools assist the author by dealing with bibliographic references or by detecting errors in spelling or even in style [Che81]. It was also at this time that Donald Knuth [Knu84] defined TEX. TEX's principal concern was typographic quality, obtained at the time by means of electrostatic printers such as Versatec, even though their resolution was only 200 dots per inch.

Alongside these tools, which were principally attractive to academic computer scientists, there began to appear professional publishing systems, which were subject to confidentiality constraints and very specific to computer or typographic hardware. The high point of this development was the IBM *Generalized Markup Language.*

Another approach was launched at the end of the 70s, in particular by B. Reid [Rei80a], who regarded the language of formatters as a high-level language describing a document in logical terms rather than as a function of the desired presentation. The role of the formatter became more important since it had to be decided, from the logical description of the document, on the page layout to adopt before producing an image on a printer. At the same period, the emergence of laser printers also stimulated the development of formatters of greater typographic quality.

An indispensable complement to a formatter is an editor allowing input and updating of the document description to be submitted to the formatter. Certain editors, originally limited to this role, later took on board formatting functions, to become editor-formatters, producing documents that are immediately printable, since they are already laid out to page. But these systems have not replaced formatters, because the formatting functions they provide correspond to those in the earliest formatters, i.e. they are low-level functions. Nonetheless, these editor-formatters enjoyed a great success, firstly on specialist word processing machines, and then as application software on micro-computers.

The latest stage in development of this type of system was based on the use of graphics screens to display an image corresponding to what would be produced on a laser printer. This is the *what you see is what you get* (WYSIWYG) approach, first of all developed on the Alto with Bravo [Lam78] and then adopted in products like the Star [HN82] or the Macintosh [Wil84]. It allows advantage to be taken of all the potential of the hardware (numerous character fonts and sizes, integrated graphics) and provides a high-level user interface.

2 What is a document?

We may define a document, circularly and somewhat factiously, to be the product of a document manipulation system. Actually the common use of the term "documents" corresponds not only to the traditional production of printers, publishers etc., but also to the papers, memos, and notes produced in offices. Indeed, in some authors' definitions, computer program code is also considered to be a special form of document. "Document" is a rather general word, and rather than attempting to provide a formal definition, that certainly would fail to fit some uses of the word, we will rely on the reader's intuitive understanding of the word.[1]

3 Why structured documents?

Of all these systems electronically manipulating documents, those with which we will be most concerned here are those which consider the document not as a stream of characters throughout which presentation information has been scattered, but as an organized structure.

In the first case (corresponding to what is called "procedural programming languages" in the program domain), we tell the system how to print the text. For example: *use character font no 3 in 16 point, move to the top of the page, right justify, print 'CHAPTER 3', advance two lines, print the chapter title, etc.* In batch systems, these instructions, or attributes, are mixed into the text itself. It may not be obvious that the same is true for many interactive systems where, although the author can say (and even see on screen) that a particular word is in italics or that a particular title (or rather the words forming the title) are in bold and right justified, he nevertheless has no automatic way of renumbering sections, of putting foreign words into italics or of handling page references.

[1] As example, the American Heritage Dictionary defines "document" as "a paper bearing evidence, proof, or information."

In the second case (corresponding to "declarative programming languages"), we no longer tell the system how to do things, but simply tell it what we want done. This means working at three levels: in the first place we have to describe the *logical structure* of the document (e.g. *a book is formed of a preface, chapters, etc. A chapter is formed of a title, sections, sub-sections, etc.*). Next we have to move from this logical structure to the corresponding *physical structure* (the printed text) (*a chapter title is right justified, in 16 point, using font times bold, preceded by the chapter number, etc.*). Systems often provide a method to handle these first two tasks. Finally, we have to apply this structure, and its page layout, to the document being written, (*this is a chapter title, here is a new section, here is a footnote, etc.*) and only this job remains to be done by the author.

Writing a structured document can then be handled, as in structured programming, in a top-down way: we start off by sketching the outline, and we postpone the typographic details of page layout for as long as possible (or even forever).

But this structural viewpoint does not only allow us to lay out documents to page and print them, but also provides real processing possibilities: given the logical structure of a document, we can build up a table of contents or an index, we can automatically number sections or notes, we can scan through the document and therefore order it in different ways, we can set up different styles of page layout without changing the text itself, we can code the document by means of different formalisms, we can send it to other people, etc. and this is all possible without the tiresome use of *search and replace* operations that generally have to be carried out in non-structured texts.

4 Contents of this book

It is the principles underlying this logical structuring of documents and the advantages that can be drawn from it that are described in this book.

1. The first two papers are in a sense a survey of what can be called document structuring:
   - in the first paper, Richard Furuta examines the concepts underlying document structures and classifies the various document models implemented in the typical systems on the basis of the degree of organization they use. He also considers models developed for the representation of various components such as mathematical expressions, tables and graphs.

- in the second paper, Vincent Quint describes the various types of software implementing structured models. The two main classes of systems are structured formatters and editors. Although the first are now well known (Scribe dates from the end of the 70s), the latter are far more recent and still raise unresolved questions. Quint considers these issues, and examines the solutions that have been proposed in various experiments. The chapter also describes the types of processing carried out by all systems, whether they are interactive or not.

2. The documents produced by these tools are not intended for purely local processing. They can be shared by many authors working on different machines, or they may be broadcast by all sorts of data transmission methods. This leads at once to the problem of standardization. We have to define a representation of documents which is comprehensible to the systems used by the different users of a single document.
   - Brian Reid analyses the problem of document communication and storage in the light of the type of structure adopted. He draws on the Kiosk system for illustrations of the points he makes.
   - Vania Joloboff describes various representations that can be used to exchange documents between systems. He is interested both in representations describing the graphics structure of a document and in those representing its logical organization, where the former allow a document to be printed or displayed while the latter also make re-processing possible. He describes the main standards used, whether *de facto* standards or standards proposed by official organizations.
3. Over and above these four basic papers, this book also includes a series of papers on more specific points, corresponding to the various talks given during this course:
   - a great deal of work has been done on the development of systems for the direct creation of documents, but much remains to be done on the retrieval of printed documents, in particular if we want to go beyond the simple recognition of characters and actually build up the structure of the documents input. Rolf Ingold describes an approach to the problem.
   - a text is more than its structural form. It also has a certain semantics, induced by the elements of its physical form (we might

call this its semiology if this term did not have certain pejorative connotations). This is why we asked a number of typographers on the one hand, and a number of linguists on the other, to describe their viewpoint on this aspect of structuring:

- Richard Southall considers, as a typographer, the various stages through which a text goes from an author to production (in its final form). Basing his presentation on the concept of *meta-design* in METAFONT for the creation of character fonts, he demonstrates the value of a *meta-design* system for typography.
- Pat Norrish studies the semantic structures of texts according to their typographic representation. She analyses more deeply two of such units: paragraphs and tables.
- Jacques Virbel starts by describing a series of hypotheses allowing us to tackle the issue of the abstract representation of textual structures, from a linguistic point of view. This allows us to handle text at a high structural level and to control its physical production.
- Saïd Tazi, after showing the limits of the hierarchical model, proposes a formalism which establishes a correspondence between the formally perceptible aspects of texts and their interpretation in terms of performative discourse acts.

Finally, the bibliographic references for all these papers have been collected together at the end of the book, followed by an index of the main systems and concepts mentioned.

Concepts and Models for Structured Documents

Richard Furuta

Department of Computer Science
and the Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742
U.S.A.

ABSTRACT
In this paper we examine several levels of representation-based models that have been developed to describe document preparation systems. At the outermost level, we examine a simple model of the representations that a document passes through in the preparation process. This model identifies three representations of the document—the document model representation, the output model representation, and the display representation. The document model presents the document as a collection of abstract (logical) objects, the output model is defined over one or more two-dimensional page spaces, and the display representation is the presentation of the document on paper or on a display screen. Using this model as a basis, we next examine models associated with these particular representations; specifically, models of the document representation, models of the components of the document, and models of the output.

1 Classification of models

A document can be modelled on many different levels. At a global level, an overview model can describe the representations that the document passes through while the document is being prepared. Additionally, such a model can describe and name the transformations that map the document between representations and between instances within the same representation.

At a finer level of detail, the different document representations can be modelled. Two representations are of particular interest. The first is the document representation that is manipulated by a system's user in specifying the structure and contents of the document. In this paper, we are concerned with a subclass of documents that is called "structured documents." Such documents are modelled as built from abstract objects—in other words, the document's logical structure is described instead of its physical makeup.[1]

[1] One description of a document by its logical structure would be to specify the document as a sequence of chapters, each consisting of a sequence of sections, each section containing a sequence of subsections, each subsection consisting of a sequence of paragraphs, each paragraph a sequence of words, and each word a sequence of characters. Contrast this description with one that describes the physical placement of components on a page.

Others have described such document descriptions as examples of "generic coding" [Gol81, Gol83].

The second representation of interest is that describing the formatted form of the document that is to be sent to an output device—in other words, the representation that is directed by the output model associated with the document. Frequently, this representation corresponds to a page description of the document.

In discussing existing document preparation systems, often it is useful to categorize the systems based on the degree of their orientation towards interactive operations. In the non-interactive group are the *Pure Formatters*,[2] which accept a document description prepared by a separate editing system and then convert that description into a formatted document suitable for display on an appropriate hardware device. Here, the editing and formatting functions are separate and follow one another in sequence. In the other group, the interactively-oriented systems, are the *Integrated Editor/Formatters*. Here, the editing functions have been merged with the formatting functions into one unified, interactive system—documents are created, viewed, and revised without leaving the editor/formatter.

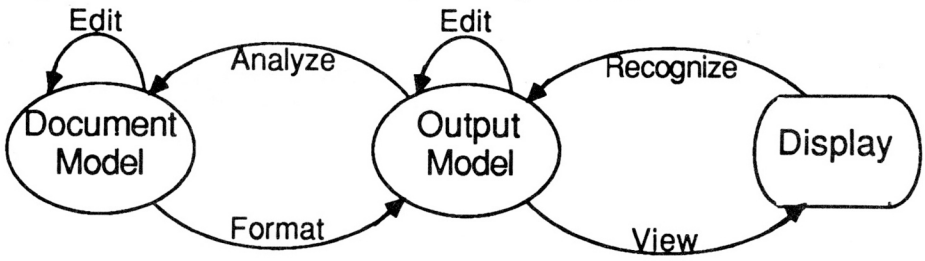
The powerful metaphor of describing the document as comprised from abstract objects is one that has been developed and applied primarily in the Pure Formatter domain [Rei80a, Gol81, Uni84, Lam85], although a topic of current interest is in extending the concept to the Integrated Editor/Formatters (see, for example, the proceedings of the EP86 [vV86] and EP88 [vV88] conferences).

A powerful metaphor applied in the Integrated Editor/Formatter domain has been to present the interactive representation of the document as being identical to the representation that would be shown if the document were to be printed on paper. In other words, the form appearing on the CRT screen has all the fonts, sizes, and spacings of the form that is printed on paper [Lam78, Sey81, SIKV82, App84]. An editor/formatter of this type is generally called a *"What You See Is What You Get" Editor/Formatter* (WYSIWYG, for short).

In the sections that follow, we first will turn our attention to describing one global document model, comparing it with several others. We then will discuss the models of the particular document representations associated with that model: the models of the document represented as abstract

[2]These definitions are based on those of an earlier survey paper [FSS82]. The interested reader will find further description and discussion in the earlier paper of many of the concepts and systems that are discussed in the present paper. See also Quint's survey in this book [Qui88].

Figure 1: The model of document processing activities



objects, the models associated with the component parts of the document represented as abstract objects, and the model of the output representation of the document. The discussion will proceed through inspection of seminal and representative systems.[3]

2 Models of the document preparation process

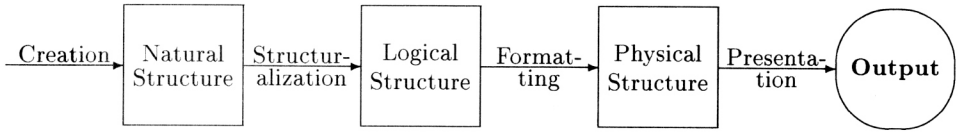
This paper's presentation is based around a simple, but useful, overall model of the activities that make up document processing. The model, adapted from that defined by Shaw [Sha80, SFS80, FSS82, KS84], is diagrammed in figure 1. Three representations of the document are identified in this model: the document expressed in terms of an abstract model, the document expressed in terms of its concrete appearance, and the concrete representation of the document projected onto a display medium.

An example may clarify the distinctions between these three representations of the document. A common document model associated with structured documents presents the document through an abstraction of its logical structure. That is to say, one such document model description would present the document as consisting of a sequence of chapters, each chapter consisting of a sequence of sections, and continuing in this fashion until the document is described in terms of its basic component parts (such as individual characters).

An output model that is appropriate for describing a page-based representation of the document will describe the document in terms of that two-dimensional page space; for example, the position of elements within that page space will be represented. An output model representation of a string of text will also include specification of the line breaks, page breaks, and hyphenations that are associated with the string of text as well as the

[3]As no attempt has been made to present a complete and comprehensive list of systems, inclusion of one system but not another similar system does not express any value judgement of the two systems' relative merits.

Figure 2: The model of text processing associated with COBATEF [PJN85]



fonts and sizes that are associated with the individual characters. In the sense that the document model presents an abstraction of the logical structure of the document, the output model can be said to present an abstraction of the physical appearance of the document.

The display representation is created by placing ink onto paper or by display on a CRT tube, to name two examples.

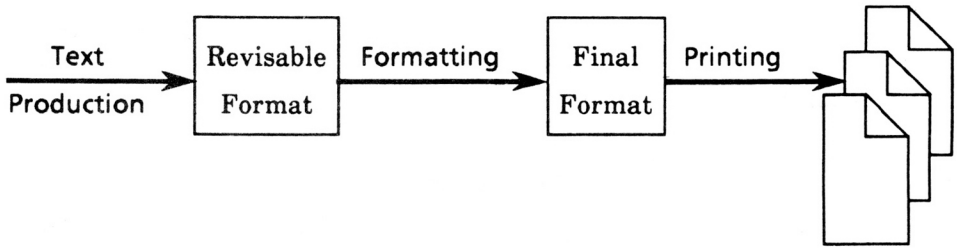
The model of the document processing activities also names the transformations between the representations. *Formatting* transforms the document model representation into an output model representation. *Viewing* transforms the output model representation into a display representation. For example, the formatting mapping may produce a device-independent representation of the output page. The viewing mapping would then translate this device-independent representation into one that incorporates the machine-dependent characteristics of a particular output device.[4]

Editing is defined to be a transformation either of document model objects into document model objects or of output model objects into output model objects. It is most common to perform editing functions on the document model—for example changes to the document's contents or its logical structure are editing operations of this type. Changes to the layout of an already formatted page (for example, rearranging the positioning of figures or tables) or modifications of character spacings on the already formatted page are editing operations involving the output model.

The particular document representations identified in this model of document processing are perhaps a bit idiosyncratic. Certainly, in categorizing the actual document representations produced by existing document preparation systems and printed on existing output devices, one finds more of a continuum of representations rather than the discrete representations identified in this model. It is not surprising, therefore, that other researchers

[4]Note that it is possible that a particular output model representation may be viewed on many different devices, with each translation carried out through a different viewing mapping to produce the particular device-dependent representation that is necessary. Similarly, one particular document model representation may be mapped to representations in many different output models through the use of different formatting mappings.

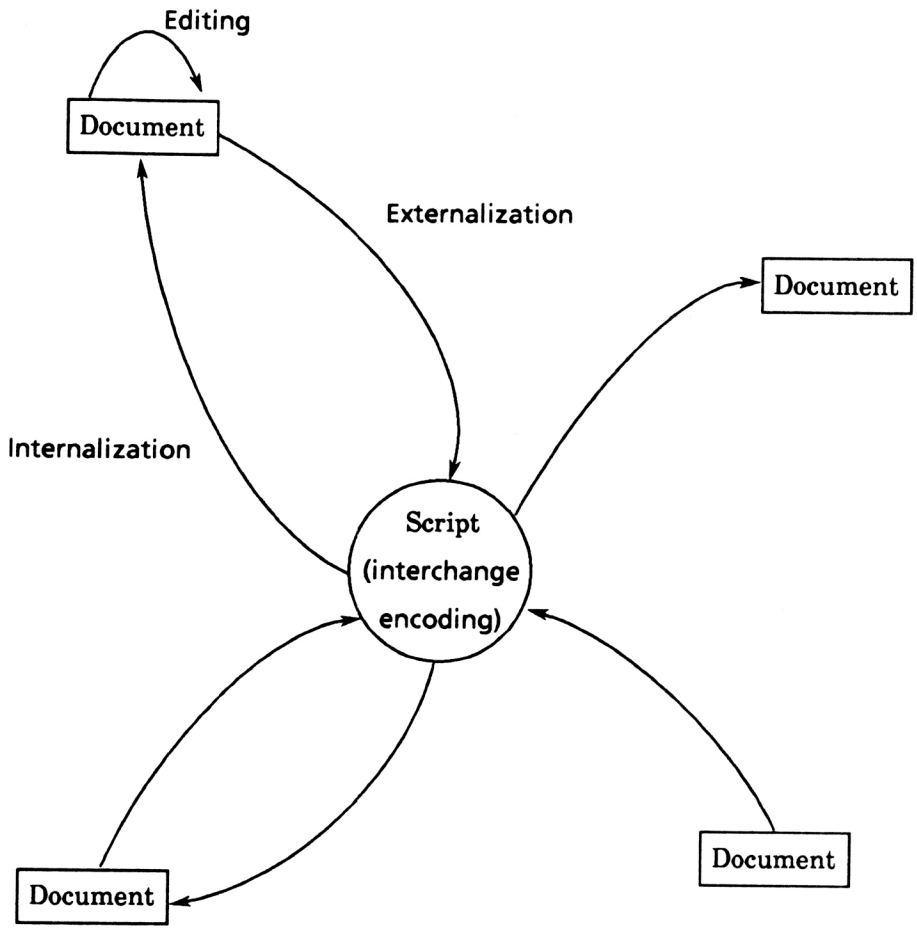
Figure 3: Joloboff's typical architecture of a document preparation system (modified from [Jol86])



have presented document models that resemble Shaw's in intent and in overall structure but that specify slightly different representations of the document. One such model, designed by Peels, Janssen, and Nawijn [PJN85], is shown in figure 2. Three representations of the document are identified in this model: the natural structure, the logical structure, and the physical structure. The natural structure representation is simply the document in machine readable form—the structuralization transformation associates a markup with the natural structure that explicitly identifies the logical structure of the text. The logical structure corresponds to our model's document model representation. Formatting converts the logical structure to a more concrete form, which is called the physical model. Like the output model representation defined in the document processing model used in this paper, the physical model defines the positioning of elements in the page space, but in addition, the physical model contains an explicit hierarchical ordering that our output model does not include: words formed from characters, lines formed from words, text-element boxes formed from lines, and pages formed from text-element boxes. Unlike this paper's model, Peels, et al.'s model is oriented to describing the representations that a particular instance of a document passes through, while this paper's model also defines the transformations that convert one instance of the document into another (the editing transformations) and the transformations that map information from the concrete to the abstract. Nevertheless, the two models are similar in intent and in terms of their view of the document preparation process.

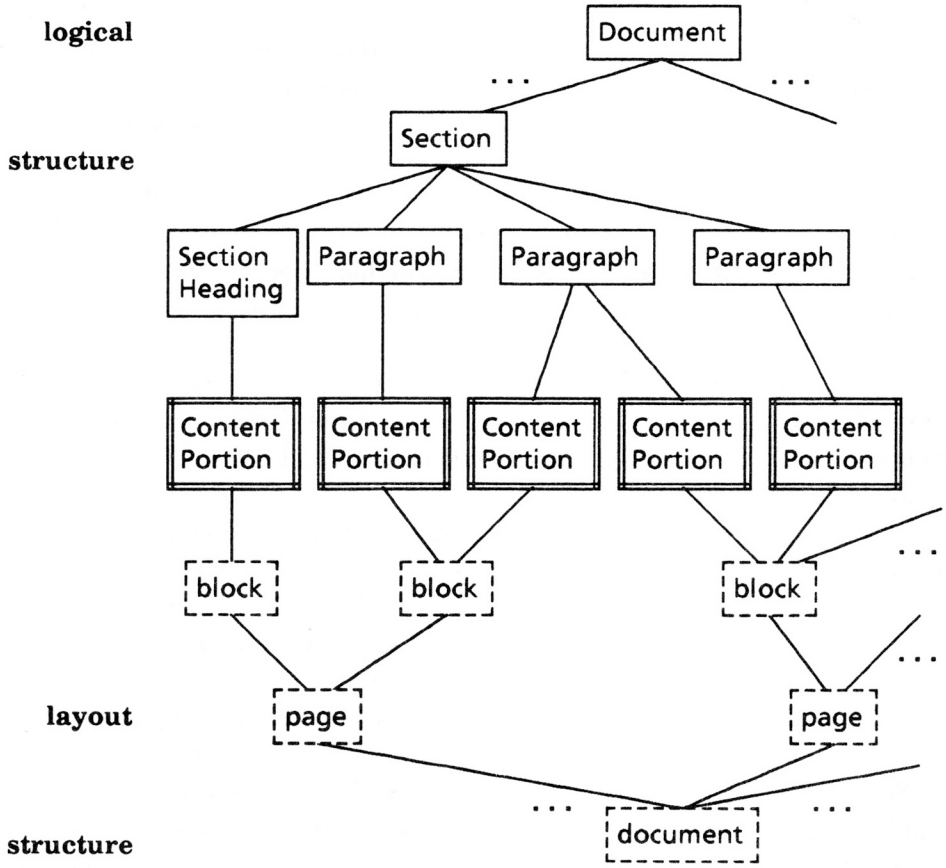
A similar model is defined by Joloboff [Jol86], which identifies two representations of the document: the revisable form and the final form (see figure 3). The process of text production creates the revisible form. The revisible form is mapped into the final form by the formatting process, and the printing process maps the final format onto the output medium.

Figure 4: The Interscript document processing model



The models described so far have been directed to describing the *process* by which a document is converted between an abstract and a concrete form. By contrast, Interscript [AHLM84] presents a model (illustrated in figure 4) that is centered around a single representation, called the *script*, that must be capable of describing virtually all editable documents, retaining all details of the documents' content, form, and structure. The script is independent of any editing system—particular editors *internalize* the script to produce a document that they are able to manipulate. When the manipulations are completed, the document is *externalized* to regenerate the script. A requirement placed on these transformations is that they exhibit "transcription fidelity"—in other words, that internalizing and externalizing

Figure 5: ODA logical and structural representations (simplification of the representation; figure modified from [ISO86a])



documents will have no noticeable effect on the document's content, form, and structure as represented in the script.

Along the same lines, the ISO ODA Draft International Standard [ISO86a] represents the content of the document and its form as separate structures. At the lowest level, the document is represented as a sequence of *content elements*—e.g., an individual character in textual material, a bit in a bitmap, and a line segment in a line drawing. Two independent tree structures are built from these most-primitive elements. The *logical structure*[5] combines the elements in successively higher-level elements representing the content of the document (e.g., paragraph, section, document). The *layout structure*[6] combines the elements into successively higher-level elements rep-

[5]The logical structure corresponds to the document model in the terminology of figure 1.
[6]The layout structure corresponds to figure 1's output model.

resenting the appearance of the document (e.g., blocks, pages, document). A set of content elements belonging to one basic logical object (paragraph, in our example) and one layout object (block, in the example) can be grouped together into a *content portion.* Figure 5 illustrates these relationships. Note that the specification of the logical structure is generally provided by the document's author and the specification of the layout is generated by the formatting process.

It is interesting to speculate on the generality of hierarchically organized object representations for modelling not only documents but also other objects in computer systems. Certainly a hierarchical document representation resembles that used by syntax-directed program editors in spirit, if not in detail.[7] Recently Notkin [Not84] and Scofield [Sco85] separately have proposed structure-oriented editing as the basis for definition of user interfaces that allow manipulation of all of the wide variety of objects found in a computer system. Notkin and Scofield each represent these objects through context-free grammar descriptions.

Notkin's design, Agave, associates a syntax-directed editor with each object.[8] Each editor is generated automatically from the syntactic description of the object augmented with action routines. Consequently the user interface for the editors is consistent as it is defined by the editor generator. Agave also allows editors to be built through the combination of existing editors. Control is transferred between editors when the cursor is moved across the boundary between the associated objects' displays.

Scofield's system, called Voodoo, presents the objects as defined syntactically, and as hierarchically related to each other. An editor is associated with a position in the object hierarchy and a particular editor can be used to edit the objects at subordinate levels of the hierarchy as well. Hence the root editor can be used to edit any object. Specialized editors, more closely tuned to the characteristics of a particular object, are associated with lower levels in the hierarchy. Scofield's design separates the editor from the objects being edited, thereby allowing definition of multiple editors that are able to alter a particular object and thereby providing multiple views of the object.[9]

[7] Certainly, as well, the kinds of objects needed to represent a document differ from those needed to represent a program in a programming language. One clear distinction is the need to allow representation of a large variety of objects in documents—a need not as strongly present when representing programs.

[8] Definition of Agave was carried out as part of the Gandalf project [MF81, MN81, AM85, ES85, Not85a, Not85b].

[9] The different editors are also related to each other in a hierarchy. A particular editor inherits the behavior of the editors that are above it in the hierarchy and can change the behavior through

In the remainder of this paper, we will turn our attention to specific models associated with documents, using the document processing model of figure 1 as our guide. We next consider models of the document representation. The later sections 4 and 5 consider models of more specific components of the document and the output models representing the page space.

3 Models of the document representation

Models that describe structured documents often represent the document as consisting of a collection of objects.[10] In evaluating the differences between models, it is frequently useful to concentrate on the definition of the objects and on the relationships that are found between objects. A basic question is to determine what comprises an object—in particular, what the lowest level "atomic" object is within the model. For example in a textually based model, the atomic object may be a string of text, intuitively representing a paragraph or a large segment of a paragraph; indeed this definition for the atomic object is one that is commonly found. Alternatively, the atomic object may be an individual character, or perhaps even a sub-part of a character.

Having determined the identity of the atomic object, the next question is to determine whether additional objects may be formed through the combination of objects and if so what relationships exist among the objects and how strongly these relationships are constrained. As example, consider a document model in which atomic objects are combined into higher level objects. This combination process continues, resulting eventually in formation of the object that represents the document as a whole. The structure of the document reflects the permissible relationships among objects—for example the relationship may be specified to result in a tree structured document or the relationship may result in a directed acyclic graph if it is permissible to include a particular object in more than a single higher level object. Within either of these structures, the relationships of objects may be further constrained. For example, if type designations are associated with each of the objects, the tree structured document will be further constrained by imposition of restrictions on the type of the component objects that can comprise a particular type of object—constraints that may be expressed, perhaps, through a grammatical specification of the relationships among object types.

its own modifications.

[10] I have also presented issues in models for structured documents in a shorter survey [Fur87a].

In general, it is often useful to describe the predominant relationship among the document objects separately from the secondary relationships. For example, although the organization of document objects into trees or other graph-like structures generally predominates, models may also include additional mechanisms that allow cross references, footnotes, and other annotation-like relationships.

An interesting distinction that can be made when comparing document models is whether a model is homogeneous or heterogeneous—in other words, are all objects formed using the same set of object relationships or can different objects reflect different organizations. For example, a heterogeneous document model might define different object relationships for material contained within an object representing tabular material and contained within an object representing textual material. A homogeneous document model would define the same relationships for all objects. Homogeneity and heterogeneity can be reflected both in the overall structure and also in the form of the atomic objects.

To consider the distinctions between document models further, in the remainder of this section we will look at the models defined by a number of representative systems.

3.1 Linear document model

One of the simplest document models is that embodied in Xerox's Bravo, Star, and Viewpoint Editor/Formatters. Bravo [Lam78] is a pioneering WYSIWYG Editor/Formatter developed in the 1970's for the Xerox Alto workstation [TMLS79].[11] Bravo documents essentially may be modelled as a sequence of paragraphs, where each paragraph is a sequence of characters. *Looks* are associated with the individual paragraphs and characters to describe their appearance. For example, a character's look describes the character's font, its size, and its baseline (allowing superscripting and subscripting). A paragraph's look describes, for example, the margins associated with the paragraph, the spacings between its lines, whether the paragraph is centered or justified, and the default character looks for characters within the paragraph.

The look mechanism was later generalized in Star [Sey81, SIKV82, Xer84b] (and in Viewpoint, Star's successor), which associated *property*

[11] The Alto is a personal workstation that was developed in 1973, incorporating an 8.5 by 11-inch bitmapped display with a resolution of about 70 pixels per inch, a typewriter keyboard, and an attached pointing device called the mouse. The concepts introduced by the software designed for the Alto have influenced the design of later systems, including Xerox's Star [Sey81, SIKV82, Xer84b], and the software developed for Apple's Macintosh, such as MacWrite [App84].

sheets with a wide range of document objects and operations. Star and Viewpoint provide a heterogeneous set of atomic objects, allowing mathematical formulæ and line drawing-oriented objects to be inserted into the sequence of objects that makes up the document.

3.2 Flat, pseudo-hierarchical document model

The Bravo document model is generalized still further by Interleaf's Office Publishing System (OPS) [Int84], a WYSIWYG Editor/Formatter. As in Bravo, the document objects found within the OPS Editor/Formatter are paragraph based. OPS uses *property sheets* to direct the formatting process. The Page Properties Sheet specifies certain properties for the entire document, for example, the margins and the appearance of the page numbering. The Component Property Sheet specifies certain properties of the individual paragraphs that describe the shape and appearance of the paragraph, for example, indentation, paragraph margins, and font.

One special object in the OPS document is the *fixed frame.* A fixed frame is a rectangular section of the document page that is considered to be one unbreakable word. It can be used to insert a formula into the middle of text or can be used to insert large diagrams into the document. The OPS editor has a separate diagramming mode that allows creation of line drawings within a fixed frame, and it also allows placement of charts within the fixed frame (the data associated with the chart is entered into the system and its appearance is selected through the property sheet mechanism).

The OPS Component Property Sheets are named, and in this way they differ in an interesting fashion from the mechanisms provided in Bravo and Star. In essence, the name defines the type of the component object and each individual component object in the document is an instance of that type. A modification of the component properties associated with an object can be applied to the type, in which case it affects all objects of that type, or it can be applied only to the specific instance. New names can be created at will and the name associated with a particular object can also be changed.

3.3 Environmental document model with limited nesting

An influential and now widespread document model is that embodied in Scribe Pure Formatter [RW80, Rei80b, Rei81a, Rei80a, Uni84]. At the outermost level, Scribe's document model represents the document as a sequence of paragraphs and named *environments.* Information to guide the document formatting process is associated with the environments. Primarily, the environments enclose one or more paragraphs although some envi-

ronments can themselves be contained within other environments. For example, an itemized list environment can itself be an element within another list environment. Some environments, such as that defining a section's title, cannot be nested and nesting is not used for identifying higher-level document structuring such as the division of a document into sections. With the limitation that some environments cannot be nested, relationships between environments are not constrained.

The primary distinction between the Scribe document model and the OPS document model is that the Scribe model associates the "properties" with sequences of paragraphs rather than with the individual paragraphs themselves. Further, Scribe's document model allows a limited form of document object nesting, which is not provided in the OPS document model.

In addition to these environmentally-based primary document structure relationships, Scribe incorporates a collection of secondary relationships. For example, Scribe provides means for associating a symbolic name with a particular element of the document (such as the number associated with a section) or with particular locations within the document (allowing, for example, identification of the page number with which the location is associated). References to the symbolic name are placed elsewhere in the document and are replaced in the printed copy the the information associated with the label. Selection of items from a separately maintained bibliography are handled using similar mechanisms.

Although Scribe is a Pure Formatter, similar models have been incorporated into interactive systems such as Etude [HIAG81b, HIAG81a, Ils80], a WYSIWYG Editor/Formatter, and Janus [CKST81, CKST82], which represents the document in two windows: a read-only window that shows the formatted form of the document and a window whose contents may be modified that shows a marked-up version of the document using a specification language that resembles Scribe's. However, the most widespread systems incorporating this model are Pure Formatters and not Integrated Editor/Formatters.[12] Incorporation of this form of document model into a WYSIWYG Editor/Formatter is not straightforward as it requires that the design provide a displayed representation that allows the system's user to locate easily the boundaries of the environments when looking at the document's display—a goal that is made harder to achieve by the restriction on the allowable displays imposed by the imposition of the WYSIWYG presentation of the document.[13] Interactive systems address this difficulty in varying

[12] A recent system of this form is LaTeX [Lam85], a macro package implemented in TeX [Knu84].
[13] I have discussed the implications of the restriction in more detail in [Fur87b].

ways. For example, OPS has addressed these issues through simplification of the document model, eliminating nesting and associating properties with the individual components of the document rather than with sequences of document components, as is found in the environmentally-based model of Scribe. In other words, OPS reduces the need to be able to detect the boundaries by simplifying the structure represented in the document model. The next system to be discussed, Tioga, addresses the problem by relaxing the constraints on the permissible interrelationships between kinds of document objects.

3.4 Unconstrained tree-like document model

Tioga is the editor and document preparation system for the Cedar programming environment [Tei84a, Tei84b, Tei85]. Cedar, developed at Xerox's Palo Alto Research Center beginning in the late 1970's and early 1980's, is based on the hardware environment provided by the Dorado computer [LCMO81, Pie83]. It serves as an example of a system that employs a tree-like document model but without constraints on the interrelationships between objects.

Tioga documents are tree structured, and data is associated with all levels of the tree. The nodes of the tree contain paragraphs of the document. The system's user can suppress display of nodes that are at more than a specified depth, thus permitting an overview of the document and quick traversal of the document. *Looks* are associated with the characters in the text and affect characteristics such as font and size. A *format* is associated with the node, and controls additional elements of the node's appearance such as horizontal and vertical spacing. Looks and formats are named and it is the *name* that is associated with the document elements, not the particulars of the look or the format. Those particulars are collected together into a *style*, which can then be shared between documents. Modification to the look and format details contained within the style, then, are reflected in the corresponding changes to the appearance of the document elements that refer to that style.

Like OPS, Tioga provides document objects that are instances of a named type. Similarly, the lowest-level objects are at about the size of a paragraph. And like OPS, the allowable relationships (sequential and hierarchical) are unconstrained.

The atomic objects in Tioga are homogeneous and textual. Some of the effects of heterogeneous atomic objects are obtained by the interesting extension to Tioga that is provided by TiogaArtwork [BS83], which extends the concept of the looks to illustrations. The appearance of TiogaArtwork illustrations are controlled by graphical style rules, called *graphical style*

sheets. TiogaArtwork uses Tioga's tree structured nodes to represent the illustration in a textual form.

3.5 Constrained, heterogeneous tree-like document model

My recent work has been centered around a heterogeneous document model that I call the **tnt** [Fur86a, Fur86b, FQA88].[14] The highest level structure of a **tnt** is an ordered tree, called the *strict tree* portion of the **tnt**. At the "leaves" of the strict tree portion are instances of the various *tree blocks.* An arbitrary structure is found within a tree block—the only requirement is that it be possible to define a representation-independent ordering among the components located within the tree block (i.e., one that holds consistently in the mapping to output model representation). The structures within the tree blocks collectively make up the *free structure* portion of the **tnt**.

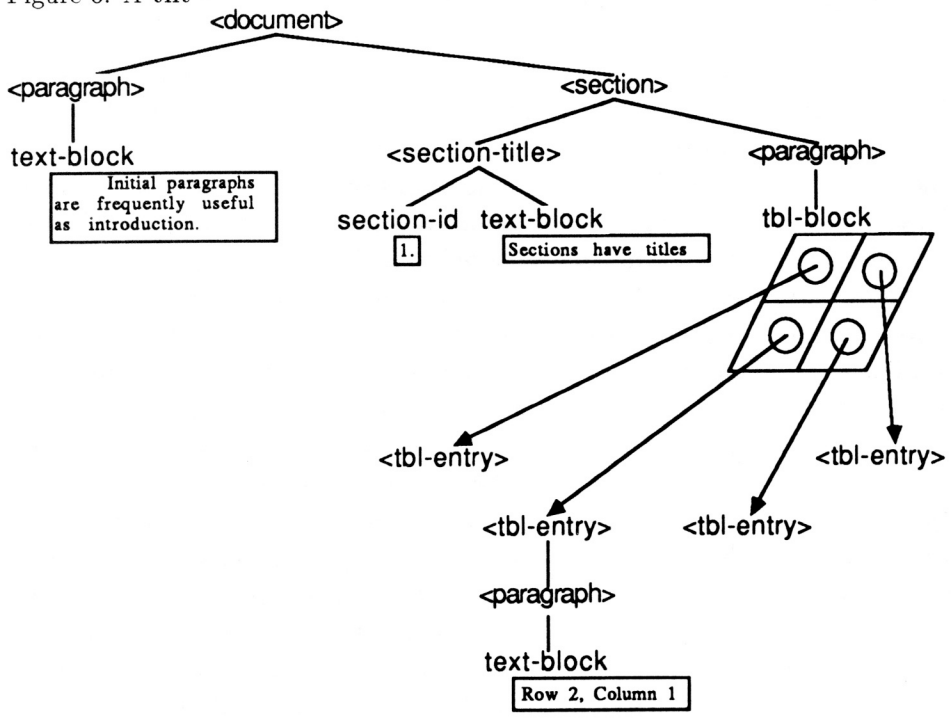
Two forms of ordering are represented by the strict tree. One is that of "containment." The document object represented by a node in the strict tree is contained within the objects that correspond to the ancestors of that node. The other form of ordering corresponds to the relative ordering of the siblings of a node—the adjacency ordering. If a node is to the left of one of its siblings, then the corresponding document object is "before" the sibling's. The relationships between objects in the tree are represented and constrained by a context-free grammar.

There is exactly one kind of structure in the strict tree portion of the **tnt** but there may be many different kinds of structures in the free structure portion. Indeed, some of the free structures may also be organized as trees. The terminating nodes of any particular free structure are defined to be either *atoms* or *transition nodes* (or both). Atoms are actual terminating points in the **tnt**. A transition node is a terminating point for the structure that contains it and also a root for an enclosed strict tree structure. Thus in traversing a **tnt** from the root downward, one encounters alternating structures—first the strict tree, then a tree block enclosing a free structure, and then perhaps a transition node that leads to another strict tree. This alternation is the reason for the choice of the term "**tnt**," which stands for "strict tree—not strict tree."

The objects represented by the **tnt** tree blocks are fairly large. For example, in a prototype implementation, three different tree block types were defined, each containing a different structure in the free structure category. These are the *text block*, representing an unbroken string of text

[14] A prototype system, called the **pedtnt** has been developed to edit the **tnt**. I have discussed issues in the design of such user interfaces in [Fur87c].

Figure 6: A **tnt**



such as a substantial part of a paragraph, the *equation block*, representing mathematical formulæ, and the *table block*, representing material organized in a tabular fashion. The free structure mechanism allows the flexibility needed to allow inclusion of a varied collection of basic document building blocks. For example, a *line-drawing block* could be defined to allow inclusion of drawings composed of line and curve segments. A *bitmap block* would allow the inclusion of arbitrary scanned images. (Models that specify these document components are described in the next section, section 4).

Figure 6 shows one particular **tnt** that illustrates these points. This **tnt** represents a document formed at the topmost level from an initial paragraph and a following section. The paragraph terminates in a text block, which is a free structure in which the components are all atoms and are organized as a string. The elements of the section's title are also instances of the text block. The final paragraph contains a table block, a free structure that is organized as a matrix and terminates in transition nodes. Each transition node serves, in turn, as the root for a new instance of the strict tree.

Looking at the **tnt** from the bottom up, it can be seen that the basic building blocks of the document are the tree blocks. These tree blocks are related to one another through the hierarchical relationships expressed by the structure of the strict tree. These hierarchical relationships differ between document classes and are what distinguish different document classes. The *same* types of tree blocks can be used in each of the different document classes. The differences between different document classes are described by the differences in the grammars that define their strict trees.

A document is modeled by one or more **tnt**s. When more than one **tnt** is present, each represents a separate *stream* in the document.[15] All objects in a **tnt** are ordered with respect to each other and generally speaking, two document objects are placed in the same stream if their ordering is maintained in all mappings to concrete representations on screen or on paper. If the ordering is not maintained, then the objects are placed in separate streams. For example, two adjacent paragraphs in the body of a document will be placed in the same stream. However, the contents of a floating figure will be placed in a separate stream from the text of the document's body as the translation to paginated output will interleave the figure with the body's contents. This interleaving will change from one translation to the next. The separate **tnt**s are related to each other through *links*, which originate in a contiguous range of nodes in some **tnt**, and terminate at the root of another **tnt**. *Visible reference points* may be associated with a link and placed within the origination range of the link. These points serve as atoms within the **tnt** and are meant to expand to be a string that identifies the target **tnt** when the originating **tnt** is displayed.

Heterogeneous tree-like models are found in the domain of syntax-directed program editors, as well. Mentor [DLM84, DKLM84] incorporates *gates*, associated with an atomic operator in the abstract syntax tree that describes the program, and *annotations*, apparently associated with arbitrary nodes in the abstract syntax tree. Gates are used to allow mixing of code fragments from different programming languages or perhaps to allow intermixing of other different kinds of internal data structures. Annotations provide a means of defining secondary relationships within the program—for example as a means for associating documentation with programming language sources or as a means to provide footnotes or other notational material in a document. It is interesting to note that motivation for the inclusion of these mechanisms seems to have been, at least in part, to al-

[15]The stream concept was inspired by Janus' galleys [CKST81, CKST82]. Janus galleys, however, contain concrete objects (i.e., already formatted objects). Streams contain abstract objects.

low facilities for manipulation of documents, rather than manipulation of programs.[16]

Finally, it is interesting to take note of a proposal by Hamlet [Ham86], which suggests the application of development methodologies taken from Software Engineering and program design to documents. An interesting concept in his proposal is the simultaneous association of multiple hierarchies with the document's leaf objects. This mechanism is suggested to allow flexible definition of the document as well as to provide a way to allow multiple views of a particular document. Hence, it can be seen that such a organization would present a homogeneous *kind* of structure for the document but that it would allow a heterogeneous representation of the structure associated with a particular document.

3.6 Truly hierarchical document model

The immediate origins of the **tnt** document model are in Shaw's earlier work [Sha80, FSS82] that models documents as abstract objects, hierarchically related to one another in a tree structure. Each higher-level abstract object is composed of lower-level objects. At the lowest level are found objects that are primitive or atomic. Objects related to each other as siblings in the tree may be either *ordered* (sequences of objects) or *unordered* (sets of objects), relative to each other. Ordered objects represent, for example, adjacent sentences, paragraphs, or perhaps rows in a table. Unordered objects represent, for example, unrelated portions of a document such as two separate appendices. Flow expressions [Sha78, Sha79] are used to express the relationships among the document objects.[17]

Shaw's model and the strict tree portion of the **tnt** are similar in many ways. In both models, lowest-level objects represent the *contents* of the document while higher-level relationships describe how the atomic objects are related. In both models, objects are related to one another by a tree structure.

A major difference between the models is that the **tnt** is a heterogeneous structure while Shaw's model is a homogeneous structure. The heterogeneity of structure in the **tnt** is provided, in part, to replace the ordered or unordered object relationships of Shaw's model. In the **tnt**'s strict tree, the

[16] Recall that Notkin's Agave [Not84] allows the definition of interacting editors, although the structures defined for each of the editors is similar. By comparison, Scofield's Voodoo [Sco85], defines a hierarchy of editors, but the editors do not interact directly.

[17] The ISO SGML standard [ISO86b] defines a document representation that shares many features with Shaw's. The SGML document representation is specified grammatically, and permits specification of ordered and unordered objects, in Shaw's terminology.

objects' relationships are always ordered. In the place of unordered tree objects, the **tnt** allows for arbitrary relationships within the free structure portion of the **tnt**. A lesser difference between the models is that Shaw's does not specify the size of the lowest-level atomic objects while the **tnt**'s strict tree terminates in the fairly large sized tree blocks.

Kimura and Shaw [KS84, Kim84, Kim86] subsequently extended and elaborated on Shaw's document model. For brevity, I will refer to this as the "KS model." As in Shaw's earlier model, the KS model represents the document as a hierarchy of abstract objects, related to each other as sets or sequences. In an extension to the earlier model, the KS model associates a unique name with each instance of abstract object in the document. This naming is used to allow specification of a full hierarchy of document object relationships—a directed acyclic graph, rather than the tree of the earlier model—in which an object may be contained within more than one parent object (*shared objects*). Unlike the earlier model, the KS model allows data to be associated with any object—not just the atomic objects (in this context, "atomic object" refers to objects that do not contain other objects—i.e., leaf objects).

The KS model also provides reference links, which are used to imply a relationship between objects (generally a relationship involving the objects' data) rather than a composition (as in the shared objects). For example, this mechanism might be used to represent the relationship between a citation and a bibliography entry. There are instances where a relationship could be described either through reference links or through shared objects, for example in describing the relationships between table rows, columns, and entries. In this sense, the shared object and reference link concepts overlap.

3.7 Relational database model

Several authors have suggested incorporating a relational model, such as that used in relational databases, into document-related systems. Macleod's Mistral/11 [Mac81] is a document retrieval system. Relations are defined and attributes associated with the relation. For example, one such relationship:

```
documents{d#, abstract}
index{d#, term}
```

defines two relations. The relation named "documents" has two attributes—an identifier that selects a particular document (`d#`) and the text of the abstract of that document. The index relation represents the entries in the document's index. Relational queries can be formulated to select those

documents that contain specific terms in the associated index and to retrieve those documents' abstracts.

In Horwitz's work, [HT85, Hor85b], the relations are associated with the nodes in the abstract-syntax trees that represent a computer program. Relations are associated with the definitions and uses of variables and procedures and are generated automatically by a syntax-directed editor rather than being specified manually. For example, one such association would associate relations with declarations of program procedures and with invocations of the program's procedures. Relational queries allow questions to be asked such as "what procedures call procedure P?" Further, since the answers to queries are also relations, they can be used to form compound queries, for example to determine all program points where variable x is defined as a function of the value of variable y. Such queries can further allow definition of alternate views of a program in which such information is updated as the program is modified.

The applicability of each of these studies to the area of structured documents is straightforward. A literal application of Macleod's work suggests means by which libraries of documents or of document fragments may be created, retrieved, and maintained. Application of these ideas to the individual objects within a document suggests means by which differing views of a document may be defined. Horwitz's work suggests mechanisms for insuring consistency in the secondary relationships present in a document (for example, to insure that the source and the destination of a cross reference are consistent or that a citation corresponds to a reference). Further, this work suggests ways in which the information associated with some of those secondary relationships may be maintained. Indeed, a database-oriented organization is commonly employeed in management of references and corresponding bibliography entries, as will be discussed later in the following section on the models associated with the individual document components.

4 Models of the components of the document

A document characteristically contains a variety of kinds of component objects—for example, it is common to find textual material intermixed with mathematical material, tables, and line drawings. In this section, we will consider the models associated with some of these document components.

The universally supplied document component object is text, generally formed from the alphabetic and numeric characters, perhaps augmented with additional application- or language-specific characters. The characters

are frequently attributed with font, face, and size specifications. The relationships between adjacent characters are one dimensional (with the proviso that superscripted or subscripted material will complicate this generalization) and so a model of the textual material as a string of attributed characters provides a good approximation.

In the discussion that follows, we will consider a number of additional document component objects, present in various existing systems. Two of the objects we will discuss, mathematical equations and tabular material, are ultimately formed from the same character set that is provided for textual material. However, the positionings between the elements are two dimensional rather than the one dimensional relationships found in the textual case. We also will discuss some component objects that incorporate symbols not found in the character set—line drawings and graphs. A final component, the bibliography for references to other works, incorporates a separate database with the source material for the individual works. Entries from the database are filtered and placed into the document as text.

4.1 Mathematical equations

Of the objects found in a document, mathematical equations are one of those that has been the most studied and many formatting systems include facilities that allow the creation and manipulation of mathematical equations. One important dimension that distinguishes the mathematical equation specification systems from each other is the amount of *understanding* of mathematical structures and their relationships that is built into the system. Two kinds of understanding are particularly relevant. One is the understanding of the *typographical representation* of mathematical objects and their relationships to each other. The other is the understanding of the *mathematical significance* of the mathematical objects and their relationships.

The level of understanding of the typographical representation of the mathematical objects varies widely in existing systems. In the early applications of computers to typesetting, conformance to the conventions of mathematical typesetting was solely the responsibility of the person entering the material into the typesetting system.[18] In **eqn**, this knowledge of typesetting conventions has been incorporated into the system itself [KC75]. TEX refines the ability of the system to represent mathematical material, incorporating much more information about the typographical representation

[18] See for example, the description of PC6 [Bar65], the discussion by Kuney, Lazorchak, and Walcavich [KLW66], and the description of Page-1 [Pie72].

of the mathematical formula's components [Knu79b, Knu84], as does the modern typesetting language provided by Scientific Typesetters, Inc. (STI) [STI83, Whi83]. In these latter systems, the mathematical components of the formula are identified to the system and it is the system rather than the user that determines the relative positioning of those elements.

Systems such as **eqn**, TEX, and STI incorporate only a limited amount of knowledge of the mathematical significance of the mathematical objects that they display and of the relationships between those objects.[19] To a certain degree, the limitations are intentional as the goal of these systems is to display the specified material by applying the conventions of mathematical style and not to modify the representation of that material. A somewhat deeper understanding of the mathematical significance of the objects is incorporated into systems such as PEN-MATH [ANP81] that contain mechanisms that allow generation of parts of the formula and provide for differing representations of the formula's components. PEN-MATH's input language is based on APL and consequently allows concise specification of mathematical structures such as arrays of mathematical objects. PEN-MATH includes an "index generator" that produces sequences of integers and also incorporates limited, ad hoc, means for performing symbolic arithmetic. The appearance of operators on the mathematical objects can be changed by altering the *looks* of the object; for example, a times b may be shown as ab, $a \cdot b$, or $a \times b$, depending on the look of the multiplication operator.

PEN-MATH's understanding of the mathematical significance of objects in a formula is largely limited to that permitting shorthand specification of the objects and permitting modification of the formatted representation of the relationships between the objects. It does not alter the relationships between the objects nor does it perform large scale modification to the objects themselves. Systems that provide these kinds of manipulations must incorporate a greater understanding of the mathematical significance of the components of a formula. Such systems have generally been those, such as MACSYMA [Mat78, Mat77], designed to perform manipulations on algebraic equations. It is interesting to note that MACSYMA also incorporates

[19] It is probably valid to say that TEX incorporates a deeper understanding than does **eqn** of the typographical relationships between mathematical objects. However, **eqn** seems to incorporate more recognition of the mathematical significance of objects than does TEX as **eqn** distinguishes function names in the formula from variable names while TEX requires explicit flagging of the function names to distinguish them. The variety of mathematical structures that can be expressed by TEX seems arbitrarily large and TEX macros can be used to directly define new ones (see $\mathcal{AMS}$-TEX [Spi80]) but the distinction made here is whether the underlying system recognizes these components of the formula directly or whether they must be marked by the user for the system's benefit.

Figure 7: TEX noad types

Noad type	Description	Example
Ordinary	an ordinary atom	x
Operator	a large operator[20]	$\sum$
Binary operator	a binary operator	$+$
Relation	a relational operator	$>$
Open	an opening symbol	$($
Close	a closing symbol	$)$
Punctuation		$,$
Radical		$\sqrt{2}$
Underlined		$\underline{x}$
Overlined		$\overline{x}$
Accented		$\hat{x}$
Vertically centered		
Inner	an enclosed part of an equation	
Fraction	a generalized fraction	$\frac{1}{2}$
Left and right	large brackets in equation	
Style	modifies equation appearance	

knowledge of the typographical representation of the mathematical objects that it manipulates—while input to the system is expressed in a manner resembling that found in programming languages such as FORTRAN and Algol, the display of the formula is in a two-dimensional form that resembles the typeset representation of the formula, to the approximation that can be provided on the standard 80 × 24 character displays that MACSYMA is designed to use.

The model of the typographical representation of mathematical objects is what is of interest in this paper. A commonly used typographical representation models a formula as a collection of nested lists of elements. A node in a list generally has information associated with it and may also contain pointers to nested lists. A physical relationship is defined between a list and the lists that are nested within it.

One sophisticated example of such a formula representation is that found in TEX. Because the design of the TEX representation illustrates the characteristics of this kind of representation, we will describe it in some detail.

A TEX formula [Knu84, Knu85b] is represented by a set of linear lists of elements. The list is called a "mlist" and its component elements "noads."

A mlist can be an element of an enclosing mlist, hence the mlists defining a formula may be considered to be related by a tree structure, or alternately the mlist may be seen to define the formula through nested lists.

Each noad contains a type and a subtype that indicates how to interpret the remainder of the node. Additionally, a noad has three fields representing the noad's contents: a "nucleus" (the central part of the expression), a "subscr" (a subscript for the nucleus), and a "supscr" (a superscript for the nucleus). The relative positioning of the noad's nucleus, subscr, and supscr may vary with differing noad types and subtypes, for example the subscr and supscr associated with a summation sign may either be placed above and below the summation sign or may be placed to the upper and lower left of the summation sign.

The characters in a TEX equation are placed into one of eight different classes: ordinary, large operator (e.g., "$\sum$"), binary operator (e.g., "+"), relation (e.g., "="), opening (e.g., "("), closing (e.g., ")"), punctuation (e.g., ","), and variable family (a special class that allows the character to change its categorization based on the context in which it is used). In addition to classifying the individual characters that make up a noad's nucleus, subscr, and supscr, TEX also classifies the noads themselves. The noad types are shown in figure 7.

The noad types are specified for two purposes. Some types are intended to convey formatting information, such as the vertically centered and inner types. Others are intended to allow representation of particular mathematical objects, for example, the radical and fraction types. The point remains, however, that regardless of how precise the character and noad classifications are, they are based on the *form* of the mathematical material and not on the *meaning* of that material.

4.2 Tabular material

A common and influential model of tables is that originated in the Unix system's **tbl** [Les76]. **tbl** represents rectangularly shaped tables that are comprised of rectangularly shaped entries. The entries can be thought of as being aligned along a matrix of rows and columns. The **tbl** table specification is divided into three parts: options, format, and data. The options change global details of the table's formatting such as whether a box is drawned around the table and around the individual entries. The format section specifies a sequence of row templates for the table, for example indicating if the entries in the associated columns are to be centered, flush left, flush right, or spanned into the space normally occupied by an adjacent en-

try (as will be described later, the global formatting information also can be changed during table specification in a limited way). Following the template is the data of the table (along with some additional description of separators between particular rows and columns of the table). The data is presented row by row, from top to bottom, and within each row in column order from left to right.[21] Limited modification of the global format is permitted within the data's specification (specifying that a line is to be drawn between two adjacent rows or columns, specifying vertical spans, etc.).

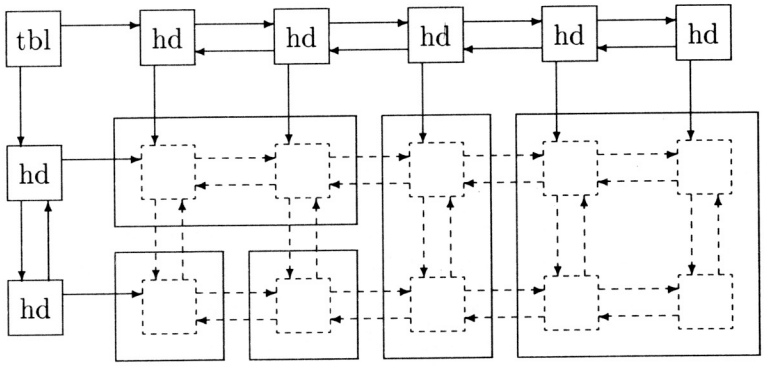
The basic **tbl** structure can be seen as a rectangular table divided into a matrix of table entries—each located at a particular row and particular column. Horizontal and vertical spanning allows a particular entry to occupy more than one row position or column position but the shape of the table entry remains rectangular. The global formatting information associated with the table can be changed between rows of the table but the changes cannot alter the number of columns or their size. Hence, even with changes to the global format, the basic underlying structure of the table—as a matrix of entries—remains intact.

Biggerstaff, et al.'s TABLE [BEF84] uses the same model of the table—rectangular and comprised of rectangular elements—and presents an interactive, hierarchical view of the table's structure. In TABLE, an extended cursor encompasses the part of the table that is being manipulated. "IN" and "OUT" function keys move this focus of attention inward or outward and operations are expressed relative to the objects encompassed by the extended cursor. In addition, rows and columns are created, deleted, resized, and spanned interactively. TABLE is presented as a "specialist editor," an editor with special knowledge of the particular complex structure that it edits and with special operations that allow manipulation of that structure. While the specialist editor approach allows investigation of the means for manipulating the specific structure, it is not clear if the approach incorporates the ability to merge the particular specialized structure being investigated with other structures, as is frequently desirable in specification of a document as a whole.

In my own **tnt**-related work [Fur86a, Fur86b], I have used the same overall model of the table's structure as **tbl** and TABLE. The primary difference is that my model separates the structure of the table potential locations for

[21] TEX's table specification mechanisms are at a lower level than **tbl**'s. The TEX mechanisms operate by aligning tabular material with other parts of the table using a template to specify the points of alignment. The resulting specification is more complex in appearance than is **tbl**'s. It is, however, possible to use the TEX mechanisms to implement a specification language that very much resembles **tbl**, as shown by LATEX's table processing facilities.

Figure 8: The structure of a specific 2 row by 5 column table

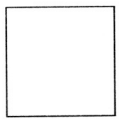


The meaning of the symbols in the above picture is:

Table head node

Row or Column head node

Skeleton node



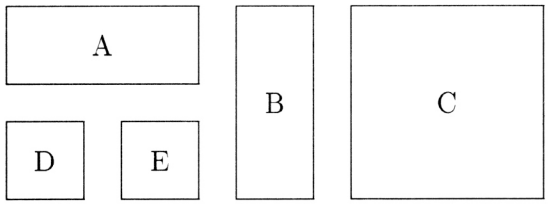
Entry node

In addition, each of the skeleton nodes contains a pointer back to its column head and row head. These pointers have been omitted to add to the clarity of the presentation.

entries (the skeleton nodes), and the actual structure of the table (the entry nodes). (The contents of the table are not represented in my model—they are to be associated with the entry nodes in implementation.) In doing so, a relationship that was expressed only implicitly in **tbl** and in TABLE is made explicit in the model.

My table structure consists of a table head node, which points to a list of row heads, and a list of column heads. Each column head points to a list of skeleton nodes. Similarly, each row head points to a list of skeleton nodes. Each skeleton node is in exactly one row list and one column list. Thus the skeleton nodes are arranged in a two-dimensional matrix and show the possible positions where table entries could be placed. Adjacent skeleton nodes may be spanned, either horizontally or vertically, which allows definition of

Figure 9: The entries corresponding the table of figure 8



entry nodes that occupy more than a single skeleton node. The constraint on spanning is that the resulting structure must be rectangular. The contents of the table are represented by entry nodes, which are terminating nodes in this structure as well as being transition nodes. Thus, the table block's termination nodes consist entirely of transition nodes. An entry node is associated with each group of spanned nodes and one is associated with each unspanned node.

Figure 8 shows a simplified view of how the structure would look for one particular table of two rows and five columns. The entry nodes defined in this example are separated out in figure 9. Note that entries "A" and "C" span two skeleton nodes horizontally, entries "B" and "C" span two vertically, and entries "D" and "E" occupy only one skeleton node and hence do not span.

A recent dissertation by Beach [Bea85, Bea86] describes the potential complexities in table layout in convincing detail. Beach suggests a framework for interactive table formatting that separates the topology of the table, described through a grid structure, from the geometry of the table, specified through mathematical constraints. The grid structure specification of the topology is similar to the matrix-oriented model seen in the other models described in this section.

4.3 Other elements

Additional kinds of document objects are found in current formatting systems, particularly in those systems associated with the Unix operating system. The additional document objects are provided in Unix through the use of preprocessors to the **troff** formatter. The languages implemented by such preprocessors are textual, not graphical, although, as will become apparent in the discussion that follows, the resulting objects defined in the document may be graphical. We have already examined two such preprocessors earlier, **tbl** and **eqn**. In this section, we will examine further some of the other kinds of objects associated with documents.

4.3.1 Line drawings

A preprocessor that deals with line drawings is named **pic** [Ker81a, Ker81b, Ker82, Ker84]. **pic** incorporates an object-based model of line drawings. The primitive objects are the box, line, arrow, circle, ellipse, arc, and B-spline. Symbolic names may be associated with objects, and the specification language uses these symbolic names to allow the description of the relative positions of the different objects. Additionally, attributes are associated with the objects, allowing specification of the size of the object, the object's orientation, and other details of the object's appearance. Objects may be collected together into a *block*, which is then treated as a single object.

IDEAL, a separate line drawing language, [VW80, VW81a, VW81b, VW82], is also available on the Unix system. IDEAL represents a line drawing through a series of simultaneous equations that specify the relations (i.e., constraints) that must hold among the points of the objects making up the drawing.

4.3.2 Graphs and charts

The Unix document processing tools' use of preprocessors to implement new graphical objects has been carried beyond a single level of application. GRAP [BK84, BK86], which implements a language for including graphs in a document, is implemented as a preprocessor to **pic**, the figure drawing language just mentioned. In GRAP, a graph is defined as a combination of the data associated with the graph and stylistic information specifying how the graph is to be presented. The stylistic information associated with the graph is defined as a set of labels, ticks, scales, line styles, and plotting symbols, expressed over a coordinate system. The data is a sequence of lines of information, each containing a sequence of coordinates and associated values. In preparing the graph for display, the stylistic elements are merged with the graph's data. Indeed if desired, the individual points of data may be shown in a graph multiple times with the application of different stylistic information resulting in a different presentation of the data points in each of their instances.

4.3.3 Bibliographies

Earlier in this paper, we encountered the definition of auxiliary representations in systems such as Scribe that allowed the inclusion of symbolically identified *citations* to bibliographic references within a document. The information associated with the references themselves is maintained in a separate

database. An *entry* in a Scribe reference database is modelled as a set of named *fields*, each with an associated value. Additionally, a unique identifier is associated with each entry. The unique identifier is used to select the entry, a tag is generated for use within the document, and the values associated with specified fields are extracted for inclusion in the document's reference list.

The Unix REFER bibliographic management system [Les78] employees a similar model of the bibliography to Scribe's and operates in a similar fashion. The unique identifier is implicitly generated in REFER, rather than explicitly specified. The citation of a REFER reference entry must include enough keywords taken from the text of the entry to uniquely identify the entry. Additionally, the REFER implementation permits use of (precomputed) inverted indices to reduce the lookup time associated with extraction of an entry from the bibliography database.

A third bibliographic management system, LaTeX's BibTeX [Lam85], employs a similar external representation of the bibliography to the Scribe representation. In fact, the external BibTeX representation is a subset of Scribe's. Hence, the overall model is still the same—a reference entry is a set of named field–value pairs. BibTeX associates additional structure with certain of the entries, particularly with the names of the author(s) of a document. The author field for a reference is a list of names. Each name contains a first part, a middle part, an auxiliary part, and a last part. The first, middle, and last parts correspond to the first, middle, and last names of an author (with a "Jr." designation considered part of the last name). The auxiliary part is used to recognize portions of a name such as the "von" in "Ludwig von Beethoven." BibTeX uses this additional information in preparing the reference list and in generating the tag that appears within the document itself—for example, a particular translation to the form to appear in the reference list might include an author's initials rather than the full name, and another might represent the author list as last name first for the first author and last name last for subsequent authors. The additional assignment of structure to the name field of the bibliography's entries increases the flexibility with which authors' names can be manipulated, but at the expense of requiring special case handling for the field.

4.3.4 Additional objects

In this section we have discussed some common component objects found in document processing systems and have described the underlying models of their representations. Additional textual and non-textual compo-

nent objects are found in some document systems including such objects as bitmapped pictures, computer program listings, material tied in an active manner to that produced by other computer programs (such as spreadsheets), and computed material—both in cross references and also in material that is to be substituted into a document to replace a placeholder (for example, in letter merging applications).

5 Models of the page space

To conclude this discussion, we will briefly consider three output models that represent the page space onto which the document is projected. The first to be considered is the TEX boxes and glue model, a model that has been quite influential on the design of other systems. The second are those based on the stencil and source imaging model, the basis of the Interpress and PostScript page description languages. And the final model will be the galley and template model of Janus—a model that is interesting because of its high level nature and because of the assumptions it embodies about the design of the formatter that produces it.

5.1 Boxes and glue

An influential model of the page space is the boxes and glue model incorporated into TEX [Knu79a, KP81, Pla81, Knu84, Knu85b]. This model represents the page through the use of nested two-dimensional *boxes* connected to each other in the horizontal or vertical dimensions by *glue*. A simple kind of box might contain an individual character. These boxes would be joined together and enclosed in a larger line box. The sequence of line boxes would then be enclosed within a page box, and so forth.

Boxes define the horizontal and vertical sizes of the object that they enclose and also contain a reference point that defines where adjacent boxes will be aligned on either the horizontal or on the vertical axis. Glue is placed between the adjacent boxes and larger boxes are formed from the horizontal or vertical sequence of boxes that results from this composition.

Glue has three attributes: a natural size, an indication of its ability to stretch, and an indication of its ability to shrink. Initially, glue is represented by space of its natural size. As lines and pages are justified, the glue will stretch or shrink in proportion to its stretchability and shrinkability. For example, justifying a line of boxes with interspersed glue can be thought of as pulling the objects in the line apart or pushing them together. Space is added to the line or removed from the line based on the stretch and shrink

factors. An appropriate assignment of glue within the line, therefore, would be to place relatively inflexible glue with a very small natural size between characters in a word (glue with little stretch or shrink that contributes little or no space to the line), somewhat more flexible glue with a natural space equal to the default word spacing between words on the line, and even more flexible glue with a natural spacing equivalent to the inter-sentence spacing after sentence ending punctuation marks. On stretching the line for justification purposes, then, little space would be added between characters, more space would be added between adjacent words, and somewhat more space would be added between adjacent sentences.

Another example showing the power of the boxes and glue model is to note that centering a line of text is simply the process of putting glue of infinite stretchability on either end of the boxes that make up the line. Any space that is to be added to the line will tend to be added to the infinitely stretchy glue, resulting in the centering of the line.

The boxes and glue output model is conceptually simple while also being expressive in terms of the textually-based pages that it can describe. As a result, the model (or variations of the model) have also been adopted by other document processing systems. Two examples of systems that use the boxes and glue model to describe their output are Etude [Sch81] (earlier mentioned in section 3.3 because of its incorporation of a Scribe-like document model into its Integrated Editor/Formatter) and the COBATEF system [PJN85], whose overall model of the document preparation process was mentioned earlier in section 2.

5.2 The stencil and source imaging model

An influential page model is that embodied in the Interpress [SR84, Men84, Xer85] and PostScript [Ado85a, Ado85b] page description languages. The two languages share a common imaging model—that described by Warnock and Wyatt [WW82]. The imaging model is centered around *stencils* and *sources*. In analogy to silk screening, the source defines "ink" that is forced through the holes in the stencil and deposited on the output medium. The "ink" may be a single color or it may be a multi-colored, two-dimensional bitmapped image. Note that characters are represented in the same way as other objects in this model—as letter-shaped holes in a stencil. The holes in the stencil may be defined in one of two ways—through the use of enclosed shapes that define the outlines of holes in the stencil (shapes) or through a matrix of binary ones and zeros where the ones indicate the location of the holes (masks).

A second level of stencil is defined as well to provide a clipping region. In other words, the clipping region defines the outlines of the display region—material falling outside of the clipping region is not displayed.

The imaging model also provides general mapping facilities for transforming both stencils and sources. For example, stencils (or sources) may be rotated, skewed, scaled, etc.

5.3 Galleys and page templates

Earlier in our discussion of systems incorporating an environmental document model with limited nesting (section 3.3), we encountered the Janus system [CKST81, CKST82], which showed a GML-like markup of its document on one window and simultaneously, a view of the document as if it were formatted on another window. Janus is also of interest here because of the high-level nature of its output model. The Janus output representation of the document employees a collection of *galleys*—in essence fragments of the document in which line breaking but not page breaking have taken place. Galleys contain information from different "streams" of the document—one galley might contain the main text of the document, for example, while another contained the footnote material. Points in the separate galleys are marked as corresponding to one another, thereby permitting footnote text to be associated with the relevant part of the main document text. Galleys are placed on the page based on the information contained in *page templates*. There may be a number of different kinds of page templates associated with a particular kind of document (e.g., title page templates, main body templates, etc.), and it is possible to associate a template with a particular page, thereby allowing the overriding of Janus' placement decisions for that one page.

In addition to noting the high-level nature of this galley- and template-based model, it is interesting to note that the model has separated the process of line breaking (i.e., forming the galleys) from the process of page breaking (i.e., placing the galley material into the templates). In TEX the process of line breaking and page breaking are carried out in tandem—the line breaks and page breaks that are computed by TEX are interrelated. In Janus, the model itself reflects a separation between the two processes.

6 Conclusion

In this paper, we have examined some of the models that are associated with structured documents. We have concentrated on the models of the

representations of the component objects of such documents and on the representations through which such documents pass from creation to display.

To conclude this examination, it is relevant to note that while we have concentrated in this paper on representations, representation models alone do not completely describe all aspects of documents and the systems that create them. An important class of distinctions may be drawn around the characteristics of the description languages that are associated with the steps in the document preparation process. As one example, the languages associated with the document model may be procedural or they may be declarative [FSS82]. Similarly, Brian Reid [Rei86b] has categorized page description languages as being either command-stream image description languages (for example, those languages associated with laser printers such as those made by Imagen and by QMS), or as being procedural page description languages (embodied by Interpress and PostScript, both of which are also postfix stack-oriented languages).

Finally, a more general, but relevant, set of models are those that attempt to model the actions of a system's user, as opposed to the topic of this paper, which has been to discuss the models associated with the system and not its users. One example of such a model is Reisner's [Rei81b], which attempts to represent the activities of a user and the sequence in which those actions are taken through description by context-free grammar productions.

Systems for the Manipulation of Structured Documents

Vincent Quint

INRIA/IMAG-LGI
BP 53X
F-38041 Grenoble Cedex, France

1 Introduction

Many computer-based document preparation systems are available today. These systems come in many different types. Some of them are interactive, others are batch processors. Certain use small personal computers, others powerful mainframes. Some simulate conventional equipment such as typewriters, others follow more abstract models. Some are more oriented towards typography, others towards the author. Document handling systems can therefore be classified by numerous criteria, those we have been considering and many others besides.

In this article, the criterion we shall use is the document model on which document handling systems are based. A possible model is one that describes documents as a string of printable characters interspersed with presentation control commands, such as line or page feeds, changes of font, of spacing, of justification, etc. Another type of model considers the logical structure of documents and, rather than describing their physical presentation, describes their organization in terms of chapters, sections, paragraphs, notes, headings, etc. It is the latter type of model which will interest us here.

We shall start by presenting a classification of document handling systems, covering every type of system, whatever the model adopted. We shall then move on to concentrate only on systems which consider the logical structure of documents. Here we shall be concerned both with interactive systems and batch processing systems, with systems that are available commercially and those that are research prototypes. The document models used will vary, but all the systems presented draw on document structure at one level or another. The aim of this article is to show what is available to users, for the various types of model.

2 A classification of document handling systems

All document handling systems currently available can be divided into four categories:

1. text editors and word processing systems,
2. formatters,
3. syntax directed editors,
4. specialized tools.

Of course, this is a fairly rough division and there are systems which have features of more than one family. Nevertheless, the four categories are sufficient to characterize the products currently being used.

2.1 Text editors and word processors

On their own, general purpose editors [MvD82] can only produce relatively rough documents. They are most commonly designed for handling programs and, for other types of documents, they can only be used to prepare work which will then be submitted to a formatter. There is, moreover, a degree of similarity between programming and the use of formatters. Editors as a class are well represented by Emacs [Sta81], which covers most of the functions generally provided by this type of tool.

Word processing systems form a family of tools better suited to document handling. They originally derived from editors, whose basic functions they have taken over, but they have additional functions allowing more elaborate presentation of documents. They allow centering and justification, and they can assign *typographic* attributes such as italics, emboldening or underlining to certain parts of a text. These systems generally handle document printing. They see to pagination, and they can also modify certain parameters such as line spacing or margins during printing.

Designed to be accessible to anybody, word processing systems are generally simple to use. But their functionality is still fairly limited, and they leave much of the work to the user as soon as the document reaches a certain level of complexity. It is the user who has to number chapters, sections, figures, etc. It is the user who has to manage references (such as *see figure n*), draw up tables of contents, tables of figures, or indexes. Above all, it is up to the user to update all these elements whenever any changes are made to the document. Again, it is the user who has to ensure uniformity of presentation and who must check that paragraphs and pages have been correctly constructed (avoiding, for example, paragraphs beginning on the

final line of a page, or ending on the first line of the next). In a nutshell, this type of system is best suited to handling simple documents.

For highly structured documents, other systems are available, in the *outliner* class. These are tools intended to assist in the organization of ideas which will later be expressed in a document. They are used in the first stage of elaboration of the document. ThinkTank [Her84] was the first system of this type. Essentially, it provides for handling a table of contents, in which the hierarchy of sections is clearly shown on screen by means of indentation. Many operations are possible on sections, at every level, such as creation, deletion, changes of level, displacement, reordering or alphabetic sorts. The user has control over the level of detail shown on screen: he can request only headings of the highest level to be shown, giving an overall view of the whole document; he can ask for headings of the first n levels to be shown for a given section, if he wants to look in detail at the structure of that section. After each section title, he can insert a certain quantity of free text which will only be displayed on request. Finally, document printing is seen to by the system, with the possibility of adding section numbers.

Outlining systems solve certain problems posed by word processing systems, but they do not provide the full range of facilities for editing and presenting the text itself. They are more concerned with the structure of the document than with its contents. They are, in fact, tools that can supplement word processors, rather than compete with them.

Finally, it is worth mentioning a recent development towards interactive systems, which has led to the emergence of systems for producing technical documents. Though they do not cover the whole complex field of technical documentation, these systems tackle certain aspects of it, and notably the insertion and handling of pictures and diagrams within documents. The Interleaf system [Mor85] is one of the first representatives of this class which has appeared on the market. It will be discussed below.

2.2 Formatters

All the preceding systems are interactive: the user sees the image of the document he is handling on screen and every modification he makes to it is immediately reflected in the display. He can therefore react rapidly if he does not obtain the results he was expecting. Formatters, on the other hand, are non-interactive tools. Jobs are submitted to them in the form of descriptions on file, they carry out the job and return the overall results after a certain time. The user cannot intervene in the execution of the process.

If the result is unsatisfactory, he has to use an editor to modify the file describing what he wants to obtain, and relaunch execution of the whole.

Though this way of working may seem cumbersome, the compensating feature is considerable processing power. Most formatters carry out operations unknown to word processing systems, notably the introduction of numbering, the compilation of contents tables or indexes, or management of references. They are also distinguished by greater typographic quality, particularly for word breaks, kerning, ligatures or the construction of lines and paragraphs. Finally, they can handle large documents, which is often difficult in interactive systems.

Two classes appear amongst recent formatters: high level formatters and low level formatters providing a macro-instruction mechanism. The first group is represented in particular by Scribe [Rei83]. The second group covers systems such as TEX [Knu84] or Troff [KLO78].

Despite the processing they carry out, and their high typographic quality, formatters are often criticized for their lack of interactivity. New tools have therefore been developed to deal with this difficulty, in part at least. These are *proofing* systems, which display an exact representation of the pages for printing on a graphics screen. This type of tool makes it possible to save some time and paper. The user can spot errors on the screen though he cannot correct them on the proof he has been shown. He has to return to the document description and find the command or the text segments to change. This limits the value of the tool.

Janus [CKST81] goes further in this direction. It was designed from the outset as a system bringing together a high-level formatter and interactive facilities. It uses two screens (hence its name), one of which is alphanumeric and displays the description of the document in the formatter language, while the other is graphic and shows the formatted image. The two screens simultaneously display the same part of the document in its two forms. Consequently, the user can rapidly find the part of his document description that generates a particular part of the formatted image. Modifications are made only on the document description and a specific command transfers their effect to the formatted image, when the user so chooses. This is not, therefore, a truly interactive system since, on the one hand, the effect of commands is delayed and on the other hand the graphic image cannot be used directly for modification. Note, however, that the user can make changes to the formatting carried out by the system, by working on the formatted image, but this is the only real form of interaction available. Janus is nonetheless a transitional system between formatters

and the interactive editors of structured documents which we consider below.

2.3 Syntax directed editors

Syntax directed editors, for their part, are generally truly interactive systems. Although they are principally intended for program handling, the principles they apply are of great value to other documents. In any case, it will become clear below that they have exerted a major influence on the design of interactive systems for document preparation.

Unlike general purpose editors described above, syntax directed editors know the language in which the program they are processing is written. Some of these editors can only handle one language. This is the case, for example, of Adele [EKMR83], for Pascal or the Cornell Program Synthesizer [TR81], at least in its first version, which was intended for a subset of PL/1. Others, on the other hand, can be parameterized. They accept a formal description of the syntax and semantics of programming languages and then handle programs written in the specified languages. This category includes Mentor [DKLM83], Pecan [Rei84] or Aloe [Med82].

Syntax directed editors use their knowledge of the syntax of the language to guide the user towards the construction of a correct program. They can build up a skeleton of the program which the user then completes. At every stage, they suggest valid instructions in each environment and check that the whole program is still consistent with the grammar of the language. In this way, they build up the program from the elements provided by the user. But they are also capable of analysis, accepting program fragments in their text form, and extracting their syntactical structure from them.

In fact, syntax directed editors are just one of a series of tools provided within a programming environment or a software engineering environment. They are therefore often accompanied by other tools used for the production of documents normally associated with programs, such as specification documents or manuals. These document production tools can be derived directly from the syntax directed editor itself, as is the case in Mentor [Mel86, Mel87]. They can also be based on the programming environment and provide a more independent tool, as is the case in the system provided by Concerto [DP86, DP87].

2.4 Specialized editors and formatters

Most of the systems described so far were designed in the first place to handle text. There are, however, often non-text elements within documents, such

as diagrams, drawings, photos, tables, mathematical or chemical formulas. This type of element raises two distinct problems: one is their inclusion within the body of the document, the other their production itself.

Depending on the type of system, the incorporation of such elements is possible in two different ways. The element to include can be created and handled by the system itself. This category includes both formatters and word processors. In the case of a formatter, the document description language is supplemented to allow the description of mathematical formulas (as in TEX [Knu84] or Eqn [KC75]), of tables (as in TEX, Tbl [Les76] or Tblp [Dew86]), of graphics (Grap [BK86]) or drawings (Pic [Ker82], Ideal [VW81a]). In the case of an interactive system, the system generally includes one (or several) specialized editors for handling a particular type of element. Word processing systems are usually limited in this area, and it is often necessary to call on technical document handling systems to obtain this type of tool.

The element to integrate can be created and handled by a specialized tool, outside the document handling system. Here again, there are both formatters and interactive systems. In the case of formatters, commonly the problem is one of inserting pictures or graphics derived from other applications (CAD systems, etc.). They sometimes make it possible to reduce or enlarge the element in order to fit it into a precise area, to conform to the format of the document. The element to include has to be represented using a standard mode, such as PostScript [Ado85a] or a graphics standard [Bon85]. When it comes to interactive systems, MacWrite [Wil84] is a good example: specialized tools allow sketching of drawings, diagrams or formulas, and MacWrite can take elements into a text and resize them. But it does not allow modifications to the contents, and will not allow insertion within the text in the middle of a line. If modifications are necessary, the appropriate tool has to be called up again and the modified element reinserted.

The means of production of these different elements vary according to the nature of the element to produce. Formatters will not accept the description of pictures or photographs, but can only integrate such elements, while interactive tools generally provide a means of creating and modifying pictures or to touch up digitized photographs.

In the graphics field (line drawings), certain formatters can be used as well as certain interactive systems [FD82]. Amongst the most common formatters, it is probably Troff which provides the most facilities in this field, with its preprocessors Pic or Grap, but then there is also Tioga [BS83]. However, interactive systems designed to illustrate documents are generally eas-

ier to use. As for other document production tools, Alto [TMLS79] showed the way with Draw [Bau78], which allows a drawing to be described as a set of basic graphics elements (rectangles, segments, circles, ellipses, curves, text, etc.) positioned in relation to each other and assigned attributes (type and weight of line, fill texture, etc.). This approach has been taken over by many other tools which stress either the drawing handling mode (MacPaint and MacDraw in particular [Wil84]), or the relation between elements [Jol83].

For the presentation of tables, formatters provide useful facilities, while interactive systems have few specifically adapted tools. Of course, graphics tools can be used for the construction of tables but not being primarily intended for this usage, they make table handling somewhat cumbersome.

In the field of mathematics, the same observations apply. While TEX, and to a lesser extent Eqn, but also Scribe and Mint [Hib83], can produce mathematical formulas of good typographical quality, the use of interactive systems raises delicate problems if they are not systems specifically designed for formula handling. On the other hand, a specially adapted interactive tool, such as Edimath [Qui83a] or the tool provided by [Lev83], makes it considerably easier to input mathematical expressions.

If formatters are handling non-textual elements, they always do so in a very structured way, with high level commands. A table is not described by giving the precise position and length of each ruled line or each character string, but by describing its organization in rows and columns, and giving the contents of each cell. In the same way, formulas are not described by means of commands governing offset, positioning or font changes, but through their mathematical structure.

3 Formatters

After this brief overview of document production systems, we shall now examine in greater detail systems where the logical structure of the documents is taken into account, and we shall start with formatters, before considering interactive systems.

As we saw before, in general terms a distinction can be made between two major classes of formatters: high-level and low level systems. High-level formatters work on the basis of a logical description of documents. The user is not required to specify the details of the presentation he wishes to obtain. He merely indicates the type of the various parts of his document (heading, summary, section, etc.) and the formatter handles its layout presentation,

which in particular makes it possible to guarantee a high degree of homogeneity in the appearance of printed documents, since elements of the same type are always presented in the same way.

Low level formatters make it possible to include commands within the document description to produce changes of font and spacing, modifications in margins, different types of justification, etc., in fact the same commands that are used in word processing systems. But these formatters in addition provide a means of specifying higher level commands, analogous to those of the preceding category. This is made possible by macro-commands which can be defined in the file containing the document itself, or in libraries. With libraries, macro-commands can be used in all documents and the formatter then comes close to the category we were considering before. LaTeX [Lam85] illustrates this approach, since its aim is to provide the functionalities of Scribe while basing itself on TeX. In the same way, the `-ms` library in Troff allows document description at a higher level than provided for by Troff alone.

3.1 Scribe

The aim of the Scribe project, developed by B. Reid at the Carnegie-Mellon University, is clearly stated in [Rei83]. The goal was to provide computer scientists with a simple and relatively specialized tool for the production of attractive and readable documents. Reid was setting out to take advantage of rapid dot-matrix printers, while requiring minimal effort from his users. He wanted, above all, to avoid their having to learn typography or a new programming language. Another objective of the project was to provide a mechanism for communicating abstract documents between research laboratories.

The principles of Scribe follow the document preparation and production methods of the traditional editorial process, which involves several professionals, each of whom plays a particular role: the author (or authors), the refereeing group, the editor, the series director or editor in chief of a review, the copy editor and the typesetter. Scribe plays the role of several of these professionals. It handles the form of the document and leaves the user responsible only for its contents.

The user works at the logical level. He describes the organization of the document, the abstract document, and states his intentions, but does not specify details of page layout or typography. All the information associated with document presentation is contained in the database on which Scribe will draw to produce the printed image. Consequently, by changing printer,

many different results can be obtained from the same document: a daisy-wheel printer does not provide the same typographic facilities as a laser printer. Furthermore, B. Reid presents his system [Rei80b] as a compiler accepting a high level language, the language in which abstract documents are described. Scribe can produce different concrete documents on different printers, in the same way as a Pascal program can give different object codes for different computers.

The document description language is a declarative language, and therefore non-procedural. The user *describes* the elements making up a document, but does not specify how to *process* them, how to print them. It is also an abstract language, as it does not refer to any particular machine or printer, but only to the logical structure of the document. This language, whose syntax is simple, is essentially used to introduce *marks* in the text of the document. These marks may delimit regions or indicate commands to send to the compiler, but the syntax is unique to each of the different functions. The regions defined by beginning and end marks are typed, and are called *environments*. An environment may concern a single character, a word, an expression, a sentence or larger elements such as a paragraph or a page. For example, there are italic, quotation, example, enumeration and description environments.

The available environments are defined in the database. That database also contains the description of the various types of document defined, each of which represents an overall environment that can be used for a document. The Scribe database provides numerous types of predefined documents, such as article, brochure, guide, letter, manual, report, thesis, etc. Other types of document can be added, as can new environments, but this is a particular task which is not within the capability of ordinary system users, and requires a good level of knowledge of typography.

Within the database, each document type and each environment is described in terms of attributes, in the form of a sequence of attribute-value pairs (see fig. 1). Scribe knows a hundred or so attributes defining the font to use, the size of characters, the positions of margins, the type of justification, the line construction mode, the vertical spacings used, line spacing, etc. When a document is being processed, formatting is determined by the values of these attributes. These values are defined first of all by the type of document, and then modified during processing as a function of the environment encountered. Environments define only a few attribute values, the values of the other attributes being inherited from the enclosing environment or the document type. Environments can be nested within each

Figure 1: An example: the itemize environment in Scribe

a) Definition of the itemized environment in the database:

```
@Define(Itemize, Break, Continue, Fill, LeftMargin +5,
        Indent -5, RightMargin -5, BlankLines break,
        Numbered <@@y[B]@, @@y[b]>, NumberLocation 1fr,
        Spacing 1, Above 0.5lines, Below 0.5lines,
        Spread 0.5 lines, Spaces compact)
```

b) Usage of the environment within a document:

```
This figure contains three parts:
@Begin(Itemize)
Definition of the environment in the database,

Usage of the environment within a document,

The formatted document.
@End(Itemize)
```

c) The formatted document:

This figure contains three parts:

- Definition of the environment in the database,
- Usage of the environment in a document,
- The formatted document.

other: it is possible, for example, to have an italic environment within an enumeration environment.

The language also provides commands making it possible to modify certain attribute values either for the whole of the document (modification commands appear at the head of the document), or for a small part of it. This makes it possible to introduce local variations with respect to the definitions in the database.

Scribe not only provides a simple and powerful means to obtain homogeneous and high quality typesetting, it also carries out certain processes on the document. In particular, it handles cross references, numbering systems (chapters, sections, pages, notes, etc.), and builds up tables of contents and indexes. It is particularly well suited to handling scientific documents, and allows each user to manage his own bibliographic database. It provides nu-

merous formats for the presentation of bibliographic references, conforming to the standards of most scientific publications.

In the same way as compilers of programming languages, Scribe provides for the possibility of separate compilation. So, for example, a large document, or a document written by several authors can be formatted in parts, e.g. each chapter independently of the others. Even under these conditions, Scribe will handle the document as a whole and, in particular, ensures that cross-references and numbering systems are consistent between the various parts.

Despite these qualities, Scribe does come up against some difficulties handling scientific documents. In particular, it is incapable, at least in its original version, of handling anything but text. It does not take account of tables, drawings or mathematical formulas. However, the commercial version of Scribe has overcome some of these defects and it, in particular, contains an extension for handling mathematical formulas. By way of example, the following formula:

$$\frac{1}{2\pi}\int_0^{\sqrt{y}}\left(\sum_{k=1}^{n}\sin^2 x_k(t)\right)f(t)dt \qquad (1)$$

can be written:

```
@Over(Num 1, Denom 2n)@~@Int(From 0, to @Sqrt(y))@~
(@Sum((From k=1, To n)@~@Sin@up[2]x@down[k](t))f#(t)dt
```

3.2 The descendents of Scribe

The concepts used in Scribe inspired the development of many formatters, but also of a few interactive systems, as we shall see later.

3.2.1 Mint

One formatter derived from Scribe, Mint, was developed by P. Hibbard at the Carnegie-Mellon University [Hib83]. The initial aim of Mint was to prepare tools for interactive document composition systems. The Mint formatter takes over the external aspects of Scribe, essentially its language, so that a Scribe user can immediately use Mint. Figure 2 gives an example of a Mint document. But the internal structure of the system is very different. It is an open structure and makes it possible to extend the formatter's processing facilities, e.g. to insert pictures within documents.

The architecture of Mint is based on four types of objects: lines, frames, galleys and pages. Lines contain a sequence of elementary composition units,

Figure 2: A document described in the Mint language

```
&readdef(paper)
&largetitle(Systems for the manipulation of structured
            documents)
&center(Vincent Quint)
&subtitle(Presentation)
Many computer-based document preparation systems are available
today.  These systems come in many different types...
...into four categories:
&begin{enumerate}
text editors and word processing systems,

formatters,

syntax directed editors,

specialized tools.
&end{enumerate}
Of course, this is a fairly rough division and...
```

such as words, but also mathematical expressions or symbols. Lines are assembled to form frames representing a continuous textual unit such as a paragraph, a heading or a figure caption. Frames are distributed through galleys, which are long rolls the width of a page, containing elements of the same type. For example, footnotes, drawings and main text all form three galleys. Finally, lines and galley frames are brought together to form pages.

The concepts of a galley and a frame are the main features allowing Mint to be extended or adapted. Frames notably make it possible to define sophisticated page layouts. Mint can also handle mathematical formulas. Formula 1 in Mint looks like this:

```
@fract(1,2n) @int(0,@sqrt(y),@paren(@sum(k=1,n,
@sup(sin,2),@sub(x,k)(t)))f(t)dt)
```

3.2.2 LaTeX

TeX is recognized as one of the best formatters from the point of view of the typographic quality of the documents it produces, but it is also often

criticized for the complexity of its document description language. This language is very rich and allows fine control of details of document presentation, but that richness also makes it difficult to master for certain users, notably those who have no typographic or programming knowledge. This defect is compensated for by a macro-instruction mechanism allowing the definition of new commands at a higher level.

It was this possibility that was put to good use by [Lam85], in the development of LaTeX. LaTeX can be regarded as a high-level formatter, very similar to Scribe, but implemented over TeX. It therefore takes advantage of the power and simplicity of the Scribe language and of the typographic quality of TeX. Like TeX, it allows mathematical formulas and tables to be described. A picture environment in addition allows the description of graphics made up of straight lines, arrows, circles, and text. Like Scribe, it handles bibliographies, tables of contents, indexes, glossaries, etc.

Although LaTeX is significantly different from TeX in the description of purely textual elements, tables and figures are handled almost as in TeX, where these objects are already described in a relatively structured way. For example, formula 1 would be written as follows in TeX:

```
{1\over 2\pi}\int_0^{\sqrt y}\bigglp\sum_{k=1} ^n
\sin^2 x_k (t)\biggrp f(t)\,dt
```

On this example, the only major differences between TeX and LaTeX are the description of the fraction, which in LaTeX would be written:

```
\frac{1}{2\pi}
```

and the description of parentheses: `\left(` instead of `\bigglp` and the same for `\right)`.

3.2.3 A few other formatters

FOAM [GW85] is a formatter heavily inspired by Scribe which uses a small microcomputer (an 8085 with 64 Kbytes of memory). Despite the constraints imposed by the hardware, the system implements the basic concept of Scribe: the document is described as a logical structure and a database contains the page layout information for the various classes of document.

COBATEF [PJN85] also works on structured documents, but is distinguished from the preceding systems principally by two factors. Firstly, it is original in that it defines a unique generic structure for all documents. It assumes that the documents it is handling can be described according to a unique structure while most formatters accept that there are several classes of document, each of which has its own structure. This undoubtedly

limits the field of application of COBATEF: for certain documents, there are structures which it would be difficult to describe using the single model accepted by the system.

The second original aspect of COBATEF is more interesting. It can recognize the structure of a text as it is input. Like the other systems, it can handle a text in which the different structure elements are indicated by explicit marks, but it also accepts text without marks in which the structure is indicated implicitly by punctuation characters, spaces, line feeds, numbers, etc. These implicit marks allow it to recognize elements such as headings, paragraphs, lists, or quotations. This kind of recognition should be compared with the work presented in [VV73].

3.3 Interactive assistance

To assist in the input of the more complex parts of documents, numerous formatters have interactive tools making it possible to build up a logical structure conforming to that handled by the formatter, while showing on screen the image the formatter will produce. These are not proofing systems, which display on screen the output of the formatter, but true interactive systems, where modifications can be made immediately on the formatted image. Such systems handling documents as a whole will be described below, and in this section we shall only consider those that are explicitly intended to do some of the work for a particular formatter.

As will have been made clear by some of the preceding examples, it is above all mathematical formulas that are difficult to describe in a structured way, in the language of formatters. This is one of the reasons why there are several interactive formula preprocessors. Another reason is that, even in formatters where documents are only slightly structured, such as in TEX, formulas are very structured. Moreover, most interactive formula handling systems have been developed for TEX or adapted to TEX.

One of the first structured formula editors was Edimath [Qui84]. Although it was developed independently of TEX, it was later supplemented with a translation program converting the formulas it produces into the TEX language [AGQ85]. This technique was taken up in Mathor [Dre86], a word processing system which also handles structured formulas and whose documents can be translated into TEX. The same type of system is being developed by the project described in [CFB86], but with more stress on the TEX orientation: the formula model is exactly that of TEX.

The advantage of all these systems is easier input of formulas, while the input and editing mode in unstructured formatters or word processors (see

in particular [Pal86]) is unsuitable for mathematical notation, which is by its nature necessarily structured.

The example below shows how a formula is input in Edimath. The left-hand column shows the sequence of user keystrokes (function keys are in italics). The middle column shows the image displayed on screen (the cursor is shown by '*'). The right-hand column simply contains comments.

fraction	$\frac{*}{}$	start of fraction and numerator
1	$\frac{1*}{}$	
end	$\frac{1}{*}$	end of numerator, start of denominator
2 π	$\frac{1}{2\pi*}$	
end	$\frac{1}{2\pi}*$	end of denominator, end of fraction
integral	$\frac{1}{2\pi}\int_{*}$	integral and start of lower limit
0	$\frac{1}{2\pi}\int_{0*}$	
end	$\frac{1}{2\pi}\int_{0}^{*}$	end of lower limit, start of upper limit
root	$\frac{1}{2\pi}\int_{0}^{\sqrt{*}}$	root
y	$\frac{1}{2\pi}\int_{0}^{\sqrt{y*}}$	
end	$\frac{1}{2\pi}\int_{0}^{\sqrt{y}*}$	end of root
end	$\frac{1}{2\pi}\int_{0}^{\sqrt{y}}*$	end of upper limit and end of integral

block (	$\frac{1}{2\pi}\int_0^{\sqrt{y}}($	start of block with opening symbol (
sum	$\frac{1}{2\pi}\int_0^{\sqrt{y}}\left(\sum_{*}\right.$	start of sum
k = 1 *end*	$\frac{1}{2\pi}\int_0^{\sqrt{y}}\left(\sum_{k=1}^{*}\right.$	end of lower expression
n *end*	$\frac{1}{2\pi}\int_0^{\sqrt{y}}\left(\sum_{k=1}^{n}*\right.$	end of upper expression and end of sum
sin *exponent*	$\frac{1}{2\pi}\int_0^{\sqrt{y}}\left(\sum_{k=1}^{n}\sin^{*}\right.$ exponent	

etc.

4 Interactive Systems

As in our discussion of formatters, we shall only present those interactive systems which take the logical aspect of documents into account, even if the structure in question is not always very marked. Although the formatters described above are mostly available commercially, the interactive systems discussed here are, with rare exceptions, research prototypes. This is because the interactive tools developed later. It was simpler, in the early days, to apply the concept of structured documents to formatters. It was only once formatters of this type had proved their worth, and the concepts of logical structure had been better understood, that the first developments towards structured interactive systems were undertaken. We shall see, moreover, that of the systems we have selected, those that are currently available on the market are the ones that are least rich in their handling of structures.

Of the systems we shall be considering, two are commercial products: Interleaf [Mor85] and Pléiade [NNC86]. Three are relatively old, since they first appeared in 1981; these are Etude [HIAG81a], PEN [ANP81] and [Wal81]. The Lilith machine gave rise to two interesting editors: Andra [GW84] and then Lara [Gut85]. We shall also consider a system derived from a programming environment: Mentor-Report [Mel86, Mel87]. Finally, we have selected all systems of this type that were presented at the EP'86

conference: Speed [CIV86], [CdVS86], pedtnt [Fur86a], [Ham86], W [Kin86] and Grif [QV86]. After a brief description of each system, we shall examine them as a whole, from the point of view of some of their common characteristics.

4.1 The systems considered

- Various versions of the WPS system (Workstation Publishing Software) [Mor85] have been proposed by Interleaf for the most common workstations (Sun, Apollo and VAXstation in particular). Numerous aspects of this system make it similar to Star [HN82] by which it has clearly been influenced. WPS brings together several functions: file management based on an analogy with an office, and interactive document handling using a WYSIWYG approach, including text, graphics and picture editing.

  The Interleaf editor considers the document as an ordered set of typed *components*. *Types* may be selected and created by the user. For example, for a letter we might use the types date, destination address, salutation, paragraph and signature. For a report, we would use the types title, author, summary, section title, sub-section title and paragraph. There is no very strong structure: a document is a simple succession of components, with each component having a type.

  The user is completely free to create new types and may insert a component of any type anywhere into a document. Types are not intended to structure documents, but more to ensure a certain homogeneity in their presentation. Each type is associated with a set of *properties* which are presentation attributes, specifying how the components of this type should be displayed and printed. The properties of a particular component can, however, be modified, in such a way that they appear different from other components of the same type.

  Most of the formatting task is carried out automatically, on the basis of properties. These are not simply associated with the types of components, but also with the whole document and pages. All the properties associated with each element are automatically applied by the editor during document handling, and the image displayed always corresponds to these properties. At any moment, the screen therefore shows a faithful image of what would be obtained on the printer, including the division of the document into pages.

  The Interleaf document editor includes a graphics editor which allows *diagrams* to be handled. A diagram may either be a free-

hand drawing or the graphic representation of numeric data. Free-hand drawing is of the structured type: a diagram is made up of a collection of basic objects which are related to each other. This approach is comparable to that in Draw [Bau78], or even more to that in [Jol83].

The base objects are straight line segments, rectangles, ellipses, polygons, curves and text. These objects have *properties* which define the weight of the lines or the fill texture. They can be grouped to form combinations which can then be handled as a whole. Each object or group of objects can undergo several operations such as a change in properties, in size, in position or in orientation. It can also be copied.

As well as a graphics editor, there is also a picture editor, to handle digitized photographs. This editor allows touching up of images with a series of different brushes, or pixel by pixel using a zoom mechanism. It also makes it possible to vary contrast within pictures or to invert them (positive-negative) by taking advantage graphically of the transfer function. Finally, images can be enlarged, reduced, rotated or displaced.

- Pléiade [NNC86] is a development of the GTX prototype [NN84]. It is clearly inspired by the Interleaf system [Mor85] whose relatively unstructured document model it adopts. It provides similar functionalities on more widely available machines. This is an easy system to use, with an interface based on icons, and hierarchical menus. It displays an image identical to what will be printed, itself handles the printing of documents. It contains a specialized editor for mathematical formulas, very similar to Edimath. It can be regarded as a WYSIWYG system which builds up the presentation of documents on the basis of typing of the elements that make it up.
- Etude [HIAG81a] is an interactive document editing and formatting system which stresses the user interface, and ease in learning and use of the tool. It works on a bit-mapped screen and sets out to display an image of documents of typographical quality. This image is built up from the structure: the document is typed and formed of components which are also typed. As in Scribe, Etude uses a database which defines the formats of the different types of elements.
- PEN [ANP81] is a system for the preparation of manuscripts containing mathematical notation. It is oriented towards input of structured

documents and is based on a formatter for printing the documents it produces. Formula handling uses a very concise notation, inspired by APL, allowing both formatting and the evaluation of formulas.

- The system described by [Wal81] has been heavily influenced by programming environments; it is chiefly intended for use with technical documents. In fact, two systems are presented: one is the ideal tool (in the author's view) for technical writers, the other a tool which has actually been implemented; the latter is a subset of the former. Documents have a very rich structure and the system can carry out powerful processing (reference searching, index compilation, etc.). The structure can nonetheless be made transparent to the user. As well as structured editing as such, the system provides tools to assist with writing, comparable to those described by [Che81], and document management tools: index, relations between documents, past history, comments, etc.
- Andra [GW84] is the document preparation system that has been developed specifically for the Lilith machine. Its design was influenced by Bravo [Lam78], whose user interface it takes over: windows, menus, mouse, etc. Andra always displays the final image of the document, though pages are not visible on the screen. The document is given a tree structure, though the user has no control over the tree, and does not see the structure. Each document is associated with a style sheet which contains the presentation attributes for the various types of elements in the document.
- Lara [Gut85] is Andra's successor. It was also developed on Lilith and solves some of the problems in Andra. It differs from it essentially on two points: logical structure and presentation. The logical structure of documents is limited to four universal levels: document, chapter, paragraph, and character string. Each level has a specific set of presentation attributes: e.g. specific top and bottom margins or running heads will be associated with a chapter; right and left margins will be associated with a paragraph, along with a form of justification and the indentation of the first line; a character string will be associated with a font or a spacing command. Another difference from Andra is that these presentation attributes are associated with the document file and are therefore specific to a particular document.
- Mentor-Report [Mel86, Mel87] is very like programming environment, since it was built up over Mentor [DKLM83]. Its intended use is in

any case heavily oriented towards software engineering. It has inherited many of the qualities of Mentor: it is programmable, extensible and open, and it will accept several formalisms. It also has many of Mentor's cumbersome characteristics, such as very definite changes of environment between text and structure handling. The extension of the system beyond Mentor does reduce these effects, since specific commands have been added to ease document handling and to make the user interface accessible to non-programmers. Mentor-Report proposes a single and general model on the basis of which every document can be constructed. It provides the chapter, section, sub-section, paragraph, annex, bibliography and other types.

- The objective of Speed [CIV86] is to combine the advantages of interactive systems and formatters. The system works on a specification of the document structure, which acts as a guide to editing and formatting. The descriptive specification of the formatting is separated from the contents of a document and carried out by a specialist user. Elements such as formulas, figures and program fragments can be included within the document. The system simultaneously displays a formatted image, giving the final appearance of a document, and a simpler image (only one font and size) on which the user can operate. A multi-task environment manages the editing process, and several formatting processes execute in parallel.
- The system proposed by [CdVS86] has not yet been implemented. The document model used is based on the concept of a class. Document input is guided by the class structure. The image is built up from the structure and several presentations are possible for a single class; the image displayed on screen is simply an *approximation* to the image that will be printed. The user interface makes learning easy and rapid. Printing takes place by way of a formatter.
- Furuta [Fur86a] describes pedtnt, the prototype of a system representing documents as abstract objects, while providing a *natural* form of handling, i.e. an interactive mode. The internal structure of documents is a hybrid: at the highest level there is a tree, whose leaves may themselves be structured, but with specific structures, that are not necessarily trees, and can vary according to the type of leaf (text, table, mathematical formula). Specific editors are responsible for handling the contents of these leaves, while a generalized editor handles trees. It seems that the use of alphanumeric terminals oriented the

choices made concerning document presentation, which the user sees in the form of templates representing all the elements that can be created or already exist. The image is therefore different from what will be printed. Printing is handled by a formatter that is independent of the editor.

- The [Ham86] system has not been implemented. It is more a collection of specifications. It aims to use software engineering techniques, notably top-down design and integrated environments, to improve document production. The stress is laid on structure which is viewed at a fairly detailed level. The presentation of documents is described by rules that are applied by the system, but the user may intervene to modify final page layout. The system generates a file of typographic commands, with printing carried out externally. The user interface shows clear signs of influence from programming environments, and may put off non-programmers.
- The W system [Kin86] is a development of the W-p prototype [Kin84]. It is based on the concept of a document class defined by a grammar. The system attempts to apply a WYSIWYG philosophy as far as possible. Reformatting is not automatic and the user decides what part of a document to redisplay after modification. Printing is handled by the system which for the moment will drive only one printer (PostScript and TEX are intended to be included in subsequent versions). Tables, pictures and formulas can be included in a document. Mathematical formulas are handled in the same way as any other element of the document. Finally, W has already been ported to different machines, from the Macintosh to the VAX.
- Grif [QVB86a] is an interactive system for the production of complex documents. It is essentially intended for handling structured documents, which may themselves contain structured objects such as tables, mathematical formulas, diagrams or programs. The approach adopted for document and object handling is similar to that used in software engineering for program editing: the document and objects are seen first of all as structures built up dynamically from generic models; the system generates the image displayed on screen on the basis of the structures. It was designed as a highly parameterizable system allowing the user to define his own types of documents and objects, as well as their appearance on screen. Dialogue style can also be adopted to the needs and wishes of individual users. Finally, it

prints documents using existing formatters (TEX, Scribe, Mint, Troff, etc.). An extension allows page layout on screen with direct output using PostScript.

4.2 Logical structure

All these systems take account of the logical structure of the documents they are handling, but they do not all take account of it in the same way, nor with the same aim, nor at the same level.

Only two systems, Lara and Mentor-Report, impose a single and general model, which all documents have to follow. This model defines the types of elements that are to make up a document and cannot be extended by the user. It is probably unsuitable and too constraining for handling non-technical documents.

In Interleaf and Pléiade, on the other hand, the end user has complete freedom to specify the structure of the document at the moment of its creation. He can create new types of elements according to his needs or desires and can articulate them as he wants. There is no generic structure. The model defines a syntax but associates no semantics with it, which limits the types of processing applicable to documents. In practice, only presentation can be based on structure, since the user provides formatting rules for each of the types it creates. Processing such as automatic numbering or the compilation of contents tables are not included. It is striking that the article by [Mor85], written using Interleaf, contains two sections with the same number.

In Andra, the structure is again relatively free, as the stylesheet associated with a document defines the element types available and their presentation rules, but lays down nothing concerning the logical structure of documents. The tree built up by the system is totally invisible to the user, which gives rise to structures which are sometimes a little odd, since they reflect the history of the editing commands used rather than the intentions of the user concerned in the organization of his documents. This defect, stressed by the users of Andra [AI84], has been corrected in Lara.

All the other systems are based on the same concept of a document: a specific structure built up according to a generic logical structure or model. This approach is also adopted in [Kim84]. Structure definition languages show differences at the syntactic level, as can be seen in the examples in figure 3, 4 and 5. These examples show how, in various systems, the generic structure of a particular class of document called *article* is described. While most systems use a BNF style of syntax, as in Speed and pedtnt, Grif has a

Figure 3: The syntax used by Speed

```
Article        = Header Body.
Header         = Title Authors.
Title          = CHAR *.
Authors        = CHAR *.
Body           = Section *.
Section        = SectionTitle Paragraph * Section *.
SectionTitle   = STRING CHAR *.
Paragraph      = Entity *.
Entity         = Text / TableEntity / List.
Text           = CHAR *.
TableEntity    = TABLE Caption.
Caption        = STRING CHAR *.
List           = Item *.
Item           = ItemMark Paragraph *.
ItemMark       = STRING.
```

somewhat richer language, in which it is possible in particular to specify the minimum and maximum cardinalities of lists: `LIST [2..*] OF (Section)` represents a sequence of at least two sections, with the star showing that there is no upper limit to the number of sections.

Though most systems use the same model, the fineness with which they view the structure can vary. For some, a paragraph is an atom (MentorReport, PEN), while others distinguish different elements in the paragraph, such as [Ham86], which distinguishes quotations, parentheses, etc., as is also the case, as we saw in the examples, for Speed and pedtnt. These systems allow very fine structures to be defined, but run the risk of being cumbersome to manipulate. This is why some of them, such as Grif and [CdVS86], provide an alternative to this refining of structure with attributes, allowing semantics to be added to certain parts of structure elements which are easier to manipulate.

For example, for a certain class of documents, we could define the attributes *language* (with values *English* and *French*) and *word type* with, for example, values *keyword*, *new concept* and *identifier*). The user can then assign the attribute *language* to certain parts of a document of this class, and the attribute *wordtype* to certain character strings. These attributes will be

Figure 4: The syntax used in pedtnt

```
<article>       = title authors <section> +
<section>       = SECTION-ID section-title <paragraph> +
                  <section> +
<paragraph>     = [text-block | <table-entity> | <list> ] +
<table-entity> = tbl-block caption
<list>          = <item> +
<item>          = ITEM-MARKER <entity> +
```

used during display or printing to select the character font and to apply the correct algorithm or dictionary for word breaks. It could also be used for other applications such as, for example, a document retrieval system.

Another relatively unusual concept in the systems examined is cross-referencing. Although some systems do mention it, this is generally the case of unimplemented systems or extensions planned for existing systems (Speed, [Wal81]). Grif uses a referencing system which allows non-hierarchical relations to be established between the different parts of a document. These references are used, most notably, to refer to all sorts of objects, such as footnotes, bibliographic quotations, figures, formulas, chapters or sections.

On the other hand, tree structures appear in practically all the systems. These structures are described at the generic level by three constructors:

an aggregate which brings together elements of a particular type (in Speed and pedtnt, it is denoted by a simple juxtaposition of symbols, but by the keywords `BEGIN` and `END` in Grif),

a list which is a sequence of elements of the same type and in variable numbers (denoted by the * sign Speed, by the * or + sign in pedtnt, and by the keyword `LIST` in Grif),

a choice which allows selection of one type of element from a limited set of possible types (denoted by the sign | in Speed and pedtnt, by the keyword `CASE OF` in Grif).

Pedtnt is distinguished from the others by its capacity to handle tabular structures in its table editor.

The use made of structure in all these systems varies considerably. In some of them, structure only allows the association of presentation rules

Figure 5: The syntax used by Grif

```
Article   = BEGIN
            Title = TEXT;
            Authors = TEXT;
            Body = Sections;
            END;
Sections  = LIST [2..*] OF (Section =
                    BEGIN
                    SectionTitle = TEXT;
                    Paragraphs = LIST OF (Paragraph);
                    Sections;
                    END
                          );
Paragraph = LIST OF (Entity = CASE OF
                    SimplerText = TEXT;
                    TableEntity = BEGIN
                                  Table;
                                  Caption = TEXT;
                                  END;
                    List = LIST [2..*] OF
                                (Item = Paragraphs);
                    END;
                    );
```

to the different elements of a document (Interleaf, Pléiade, Etude, Andra, Lara). For others, it also guides the user through the construction of a document conforming to a model ([Ham86], [CdVS86], W). Some systems also allow automatic numbering based on the structure (Speed, pedtnt and Mentor-Report). Finally, three systems include more elaborate treatment, notably cross-references and indexes (PEN, [Wal81], Grif).

4.3 Document images

The image presented on the screen during editing, like the image that is printed at the end of processing, is built up automatically from the logical structure of the document. This is generally based on presentation rules specifying the way in which each element type in a document structure is to

be displayed or printed. This is a principle with no exceptions in the field of structured document editors.

The way in which presentation rules are specified varies from one system to another. Some of them, above all the least structured, work only in interactive mode. In Interleaf, for example, properties (as the system's presentation rules are called) are defined by means of *property sheets*, in much the same way as in Star [SIKV82]. Property sheets are interactive forms in which all the parameters accessible to a user appear. The user can in this way define new properties or modify existing properties. There are three sets of different properties: for a document, for pages and for components. The properties associated with a document are limited. They only specify whether word breaks are to occur at the end of lines and on which printer the document is to be produced. Page properties are richer. They describe the format and size of pages: orientation, height, width, right and left margins, top and bottom margins, position and style of page numbers, text and position of headers or footers. These page properties can vary in the course of a document.

Components also have a set of properties defining not only their name, but also the way they are formatted. These properties are divided into three groups.

1. The first group indicates how the text of a component is to be divided up into lines:

   margins: space above the first line of the component, below the last line, to its right and to its left,

   indentation of the first line,

   line spacing: the vertical distance between two successive lines,

   line justification mode: ragged left, ragged right, centred, justified,

   typographic attributes of characters: style, size, weight, roman or italic,

   hyphenation: enabled or disabled.

2. The second group of properties of components concerns tabs, their position (distance from the left margin) and their type. If a line includes tab characters, character strings delimited by these characters will be positioned with respect to the successive tabs, depending on the type of tab used:

   left: the start of a character string is placed on the tab marker,

right: the end of the character string is positioned on the tab marker,

centred: the centre of the character string is placed on the tab marker,

decimal: the decimal point in numbers, or their last figure if there is no point, is placed on the tab marker.

3. The final group of properties of components concerns their position in the page:

new page: whether or not the component starts at the top of a new page.

widow control: the minimum number ν of lines at the start of the component which must appear within a single page; if it is not possible to position the ν lines at the bottom of a page, the component must start at the top of the next page.

orphan control: minimum number of lines from the end of the component which can be separated from the rest of the component by a page break,

break: shows whether a page break can take place within a component,

page break: indicates whether a page break should take place after the component.

All the properties associated with each element are automatically applied by the editor during document handling, and the image displayed corresponds at all times to these properties. At any moment, therefore, the screen gives a faithful image of what will be obtained on the printer, including the division of the document into pages. As properties are associated with pages and components in each document, they are part of the document and apply during editing as well as during printing.

Speed proceeds differently for the specification of document presentation. All the presentation rules are defined by an *attribute table* which contains, for each type of element defined in the generic structure of the class, a set of characteristic formatting attributes of this element type. Figure 6 contains an extract from a possible attribute table for documents in the *article* class defined above.

A question mark is used for attributes whose value can be changed by the user, with the value after the question mark being the default value. Note that not all the attributes are defined for each type of element. This is due to the inheritance mechanism which assigns to each attribute the value

Figure 6: Part of a Speed attribute table

```
Article      : FontFamily = ? (Times),
               MainTextFontShape = ? (Roman),
               MainTextFontSize = ? (10),
               MainTextLineSpacing = ? (13),
               MainHeadingFontShape = ? (Bold),
               HeadingFontShape = ? (Italic),
               TextAreaWidth = ? (312),
               TextAreaHeight = ? (528),
               AbsoluteTopMargin = ? (100),
               LeftMargin = ? (100),
               RightMargin = LeftMargin + TextAreaWidth,
               Language = French,
               PAGINATE (AbsoluteTopMargin, TextAreaHeight);
Header       : LeftMargin = ? (LeftMargin + 36),
               FormattingMode = LeftAdjusted,
               MaxVertSpace = ? (182-26);
Title        : FontShape = ? (Italic),
               FontSize = ? (14),
               LineSpacing = ? (17),
               TopSpace = ? (4 * LineSpacing),
               BREAK (LeftMargin, RightMargin, FormattingMode,
                      LineSpacing);
Body         : FontShape = MainTextFontShape,
               FontSize = MainTextFontSize,
               LineSpacing = MainTextLineSpacing,
               FormattingMode = Justified;
Section      : SectionNumber = COUNTIN(Article);
SectionTitle : FontShape = HeadingFontShape,
               BuiltString = SectionNumber & '.',
               BREAK(LeftMargin, RightMargin, FormattingMode,
                     LineSpacing);
Paragraph    : ParagraphNumber = COUNTIN(Section),
               Indentation = IF ParagraphNumber=0 THEN 0
                             ELSE em(FontSize),
               JUSTIFY(LeftMargin, RightMargin, Indentation,
                       FormattingMode, LineSpacing, Language);
```

of the same attribute in the element one level above, if the attribute is not defined explicitly for this type of element.

Thanks to attribute tables, the editor can automatically build up an image of the document. The user can also modify all the attributes which are modifiable, and can do this within the document itself, or in a separate file.

In Grif, the appearance of a class of documents is specified, as in Speed, before creation of the document, using *presentation schemas*, which correspond to attribute tables. These schemas are described in abstract terms making the description independent of the display or printer device used. So, for example, font size (`size`) is defined in relative terms (1 = very small, 2 = small, 3 = medium, etc.) and sizes or distances are expressed in units associated with the size of characters (1 = character height). A possible presentation schema (or rather an extract of one) for *articles* is shown in figure 7.

In most systems, the box concept is used very heavily for the construction of displayed or printed pictures, but only W and Grif make it directly accessible within presentation rules, which allow the relative positions of different elements making up a document to be specified. In all other systems, such as Interleaf and Speed, presentation rules are expressed in terms of conventional typographic attributes: font, character style, margins, indents, centering, linelength, etc.

In Grif, a box is associated with each element type defined in the generic structure and the presentation rules show relations between the dimensions and positions of boxes for the different element types. These relations are respected during document editing. Boxes are used systematically. They describe both structure elements and presentation elements such as lines built for display on screen.

For certain systems, the WYSIWYG principle is applied fully, and each document has only one image, which is the same on the editing screen and on paper. This is notably the case in Interleaf, Etude, Lara and W. Other systems consider that several images are possible for a single document, and that changes in presentation should take place only by changing presentation rules applied to the document and not by modifying the document itself. This is the case of [CdVS86], Speed, pedtnt, Andra and Grif.

The question of pages is handled in different ways in all these systems. Some of them display pages on screen; these are the systems which closely associate editing and formatting, such as Interleaf, Speed, Pléiade, [Ham86] or Etude. Others, on the other hand, give a fairly remote image of what will be printed, which is notably the case in Mentor-Report and pedtnt. Grif is somewhere between these two extremes and ignores pages if a division into pages poses more problems than it solves while still guaranteeing a good quality of the image displayed.

Many systems have a mechanism of views allowing the same document to be displayed in different ways. Neither Andra, Lara nor PEN have, however,

Figure 7: Presentation rules in Grif

```
COUNTER          SectionCounter: RANK OF Section;
DEFAULT          BEGIN
                 Size: Enclosing = ;
                 Width: Enclosing . Width;
                 HorizPos: Left = Enclosing . Left;
                 Height: Enclosed . Height;
                 VertPos: Top = Previous . Bottom;
                 Justify: Enclosing =;
                 END;
Article        : BEGIN
                 Size: 2;
                 Width: Enclosing . Width - 6;
                 HorizPos: Left = Enclosing . Left + 3;
                 VertPos: Top = Enclosing . Top + 2;
                 Justify: Yes;
                 END;
Title          : BEGIN
                 Size: 4;
                 Width: Enclosing . Width - 4;
                 HorizPos: Left = Enclosing . Left + 4;
                 VertPos: Top = Enclosing . Top + 4;
                 Line (Left);
                 Justify: No;
                 END;
Body           : VertPos: Top = Header . Bottom + 2.5;
SectionTitle   : BEGIN
                 Size: Enclosing +1 ;
                 VertPos: Top = Enclosing . Top;
                 Create (SectionNumber);
                 Line (Left);
                 Justify: No;
                 END;
SectionNumber: Content: (Value(SectionCounter, Arabic) Text '. ');
Paragraph      : BEGIN
                 IF First THEN Indent: 0;
                 IF NOT First THEN Indent: 1;
                 VertPos: Top = Previous . Bottom + .5;
                 Line (Left);
                 END;
```

such a facility. Certain systems predefine a view showing the document skeleton (Pléiade, pedtnt, W, Mentor-Report, [Wal81]). Others, such as Grif, provide a general mechanism allowing a definition, within presentation rules, of the appearance and contents of any number of views. These can be used, to show, for example, the contents table at the same time as the text

as a whole, or the English part of a bilingual document, or only the notes that are scattered throughout the document.

Although these views are a way of displaying document structure, other means have been proposed in most systems to show the logical structure. Certain systems display the path through the tree from the root to the current element (Pléiade, pedtnt), others systematically show element types in a specific zone, besides the document image (Etude) or within the document image itself (pedtnt). Several systems (Speed, W, pedtnt, Grif), allow the addition of supplementary elements in document presentation, e.g. a word such as *summary* at the top of the summary, or a character (asterisk, hyphen, bullet, etc.) at the beginning of each element in an enumerated list, or a rule to separate two elements. Note that Speed and pedtnt specify these elements in the generic structure itself (e.g. `ItemMark` in Speed, figure 3 and `ITEM-MARKER` in pedtnt, figure 4), while Grif handles them only at the presentation level.

4.4 Included objects

For certain systems, the only possible contents of documents is text ([Ham86], [CdVS86], [Wal81], Etude, Lara). Andra allows the insertion of pictures within documents, although the pictures are not handled by the editor, but merely inserted when the document is printed. Most other systems allow the inclusion of elements of various types, and handle these objects with specific tools which are part of the editor (pedtnt, Interleaf, Pléiade, PEN, extension planned for Speed). Grif and W handle mathematical formulas in the same way as the rest of the document structure, but only Grif provides a homogeneous way of handling all sorts of objects, definable by the users, with the same tools as for documents themselves: generic structures and presentation rules. This is in part because of the box-based model it has adopted for presentation. The concept of a box allows description both of text laid out in lines and mathematical formulas or tables.

For example, in the generic structures shown in figure 3 and 4, the words `TABLE` for Speed and `tbl-block` for pedtnt identify objects known to the system, and which are handled by a specific editor. On the other hand, in Grif, the word `Table` indicates another generic structure defined in the same way as for articles. This allows the editor to be adapted to different types of objects, in the same way as it is adapted to different types of documents.

Mentor-Report is somewhat special, since like Grif, it has a general mechanism to handle several different types of symbolism within the same

document, but its presentation methods are too limited to display two-dimensional objects such as formulas, tables or figures.

4.5 Interaction mode

User interfaces seem to reflect the source of inspiration of the system designers. Those most directly influenced by programming environments maintain a user dialogue similar to that of programmers, with explicit tree handling commands (Mentor-Report, [Ham86], [Wal81]). It is these systems that use a pre-existing text editor (e.g. Emacs [Sta81]), to handle structure atoms. The others are more oriented towards interfaces based on menus and on a mouse, and incorporate word processing more fully.

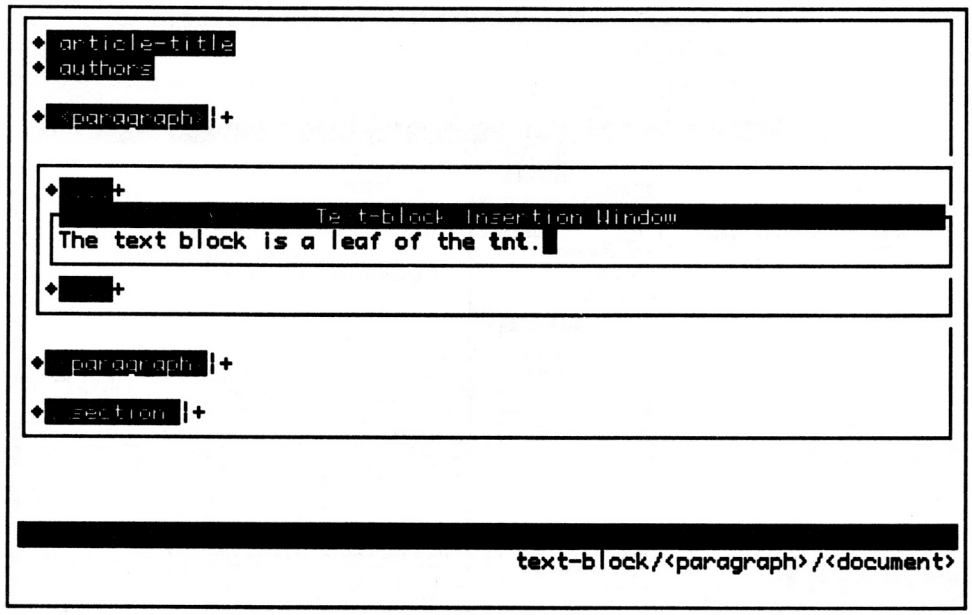
The mode of interaction also varies according to the type of image displayed and the type of structure adopted for documents. With systems such as Interleaf or Pléiade, the user can create an element of any type at any point in a document. He can also change the type or presentation attributes of any element.

When building up an Interleaf document, the user can create new types from scratch, with all their properties, as and when they are needed. Components are then created that conform to these types. But the most effective method is to create particular documents, which are skeletons of typical documents, containing at least one component of every possible type. For example, we could have a letter skeleton, a report skeleton, etc. Though it takes place interactively, the graphic design of a document can therefore be separated from the input of its contents. The creation of a document then begins by copying the skeleton document corresponding to the document to create. Empty components then have to be filled by typing their text, and new components have to be created using the predefined types. The structure of the document defined by the skeleton only provides a suggested approach: the user can easily create new types, modify the properties of existing types, eliminate components that were in skeleton, or change the order in which they appear.

On the other hand, in systems where documents are defined by a generic structure, the user has less freedom of action: he can only create, modify, or eliminate elements in a way that conforms with the generic structure of the model. The user is generally guided by the system, which is based on the model. This ensures that any document constructed conforms to the model.

A number of techniques are used to guide the user during the construction of a correct document. The pedtnt system automatically generates *buttons* for all elements the generic structure allows to be created. These buttons

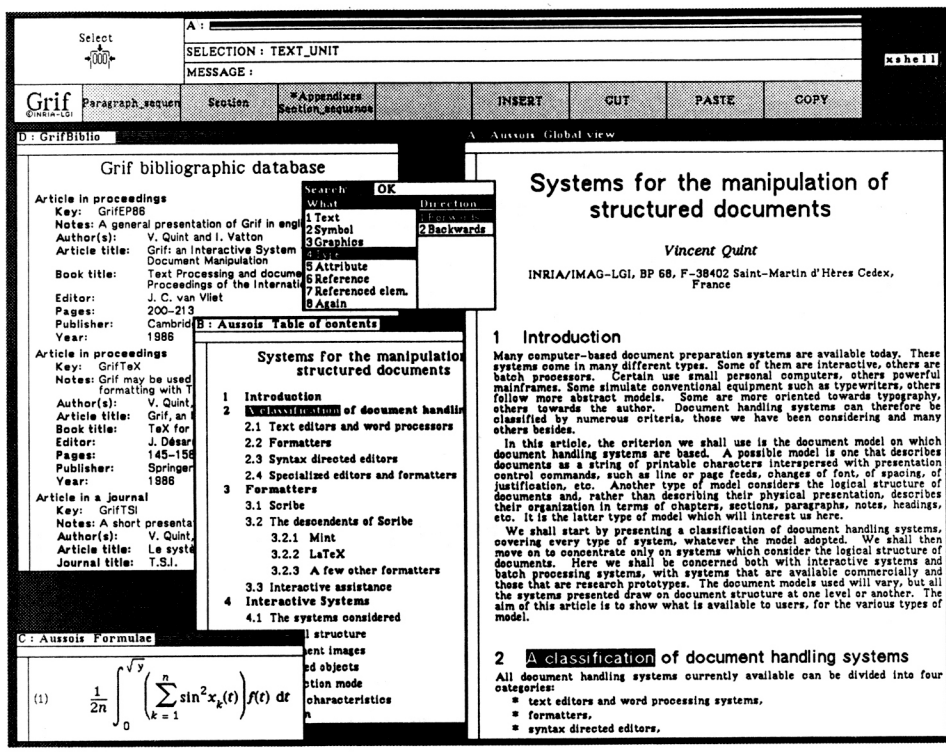
Figure 8: The screen for the pedtnt prototype



take the form of the element name that can be created, displayed in reverse video (see figure 8). The cursor simply has to be moved to one of these buttons and the Create command activated to create an occurrence of the element. The new element is then displayed with buttons showing possible neighbours.

Grif, on the other hand, does not introduce additional information in the document image (see figure 9). It uses a specific window (at the top of the screen in the figure) to display the types of elements neighbouring the selected part (that part is determined by the mouse and is displayed in reverse video) and the types of those which can be created. The user simply has to select one of these grey boxes using the mouse to create an element of the type desired. The command `INSERT` also allows new elements to be created: it provides a menu of elements that can be created after (or before) the selected part, as determined by the generic structure. When a new element is created, the elements it should contain are also created, but left empty, and displayed in the form of grey rectangles with the size and location defined by presentation rules. If the grey rectangles are not sufficiently informative, the user can select them with the mouse and their type will be displayed at the top of the screen.

Figure 9: The Grif screen



The degree of interactivity allowed can vary significantly from one system to the other. Some of them try to reflect every action of the user immediately on the screen, including typing of a single text character (Etude, Pléiade, Andra, Lara, Grif), while others only reformat the document image under certain conditions, such as an explicit command from the user, or on completion of input of an element (pedtnt, Mentor-Report).

Not all the systems attach the same importance to the user interface. Etude is particularly strong on this point: on-line help and a mechanism to undo the most recent command, for example, are available at any time, whatever the context. Pléiade and [CdVS86] also adopt this approach, though in a less systematic way.

4.6 Other characteristics

Two propositions ([Ham86] and [Wal81]) are based on the addition of the writing assistance tools already available in certain formatters to a structured editor: spell checking, style checking, synonym dictionaries, etc.

Though there are undoubted advantages in these tools, they do not seem to have been implemented in a structured document editor.

The printing mode for documents varies from system to system. Those which are most concerned with formatting, and mainly adopt a WYSIWYG approach, themselves handle document printing (Speed, Interleaf, Pléiade, W, Etude, Andra, Lara). In Pléiade, some concessions are made on formatting quality: the algorithms used are the result of a compromise between efficiency (formatting takes place in real time on screen) and the quality of the results obtained. Most systems use formatters. [CdVS86] proposes a general mechanism comparable to that used in Grif [QVB86b], allowing access to any formatter. [Wal81] uses Scribe. Mentor-Report offers a choice between Compose, Troff and TEX. In the long term, W will use TEX. Pedtnt aims at TEX, Scribe and Troff.

None of these systems conforms to a document representation standard. The document model they mostly use is not unlike SGML [ISO86b] or the logical part of ODA [Hor85a, ISO86a]. However, it will not be possible to align systems like Pléiade, Interleaf or Andra with standards, since they have no generic logical structure. Only ODA could possibly be used, applying only the *layout* part, but this would rule out any editing of standardized documents.

5 Conclusion

Whether in interactive systems or in formatters, document structuring provides the user with great power of expression. He can concentrate on the contents and organization of his document, without thinking about its layout. But a major difference between these two categories of systems, apart from the way in which they are used, is the fact that interactive systems can guide the user and put forward suggestions for possible structures in the current environment, ensuring that the document produced will be correct, at least from the point of view of its generic structure. This is one of the reasons for encouraging the development of interactive systems.

It also needs to be stressed that document structuring, in any type of system, has further advantages, which are not concerned with creation and editing alone. Because of the high level of abstraction of the model used, several types of processing can be applied to structured documents. The information necessary to a document retrieval system can be extracted automatically, without going through a manual indexing stage. In general, databases can work efficiently on such documents.

Another category of applications concern formatting or phototypesetting. Thanks to structured document production systems, it becomes possible to put a single document through different formatters without manual intervention (see for example [QVB86b] or [And87]). This provides a first step towards the development of an electronic document processing system for every stage in the editorial process.

This development will be encouraged by the convergence between interactive systems and formatters. Many prototypes of interactive editors for structured documents already produce output compatible with the best formatters, but we are also beginning to see the emergence of increasingly complete interactive environments based around formatters, and notably around TEX. The specifications for VORTEX [CCHM86] are a good example.

Document Representation: Concepts and standards

Vania Joloboff

BULL Research Centre
c/o INRIA, Route des Lucioles
Sophia-Antipolis
06565 Valbonne Cedex, France

ABSTRACT:
This article examines the problem of document representation in computer systems for printing, editing, or interchange among heterogeneous systems. After a discussion of the various possibilities for defining documentation representation formalisms, it considers a number of standard representations typical of their class: page description languages, SGML, Interscript, ODA.

1 Introduction

Document interchange between sites has become a major problem: more users would like to be able to communicate with documents in a structured form, whether it is a page structure or a logical structure, and take advantage as far as possible of the available means of communication. Unfortunately, the various computer systems used are often incompatible, as they do not 'understand' the representation adopted by the systems with which they are communicating.

A document may exist in a document preparation system under different forms: the paper sheets from the output device, the text displayed on the terminal screen, but also through more abstract representations, such as those existing in the computer memory or that used for storage on a magnetic device.

Any *representation* internal to the system is a sequence of digital values which makes sense only through some *interpretation* made by the software handling the representation. This interpretation is dictated by a *formalism* providing meaning for a series of values. Some software is able to handle a particular representation if it has been coded according to the formalism that defines the representation. It is usually said that a representation follows some *format*, meaning that the representation conforms to the syntax defined by the conceptual formalism, that is, the representation has the delimiters, the block structure, or other idiosyncrasies, typical of the formalism. Yet

both terms (formats and formalisms) have a different meaning, they are often confused.

In this paper, we shall be interested only in durable representations; i.e. those which can be stored on devices or transmitted across communication networks. Until a single style of representation emerges (if ever), the current systems impose their own formalism, even though manufacturers and, above all, standards organizations are attempting to prevent the proliferation of representations by imposing standards.

In order to understand the main representations being used in current systems, and those that are being developed by standards organizations, they are considered in the context of their aims and of their historical backgrounds. Next section traces the evolution of document preparation systems and distinguishes the three generations of systems that have successively appeared. The first generation is history now, the second is being marketed today, and the third is beginning to appear in a number of experimental systems, some of which are entering the market now. The following sections of this article examine a number of different representations.

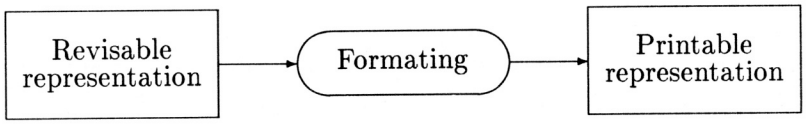
2 Evolution of systems

Document preparation systems began to appear with the first printers which were capable of producing documents with a quality equivalent to that of a typewriter. Clearly, the early goal was not to produce documents whose quality was comparable to professionally produced documents, but to guide the operations of the printer terminal and to simplify the author's task.

Printer terminals of the time (1960's) would accept commands, typically the first 32 ASCII characters, and, control sequences, which produced movements in the printer head, or allowed underlining, accented characters, etc. In general terms, the essential output from these systems was lines and pages produced from a file in which 'end of line' characters were not significant. Typography was basically the same as a typewriter's, but lines were justified, using additional spaces.

In such a system, the number of commands were limited, and they were all concerned with the layout representation of the document, not with its logical structure. One of the earliest and most representative of this kind, RUNOFF [Sal65], is historically important as its main features reappear in modern systems: the user creates a file in which the text to be printed is interspersed with commands that are interpreted by the software producing the physical document. The system distinguishes commands from text by

Figure 1: Revisable and printable representations



a special markup convention (in RUNOFF, a line beginning with the dot character). For this reason, command languages in these systems are often called markup languages.

Even in these early systems, the need was felt to be able to to print the same document on two printers whose commands are different, even though their quality may be similar. This requires a device-independent printable form. If this printable form of the document exist, it may be stored on a magnetic device, avoiding the requirement to reprocess the initial document when a new copy is needed.

The typical architecture of a document preparation system (figure 1) therefore has two forms of representation, one input to the formatter, the text produced by the author called a revisable form, and the other output to the peripheral, called the final or printable form. An author types text in (for example a `.tex` file in TEX [Knu79a], this file is analysed producing error messages and an output file (the TEX `.dvi` file) as independent as possible of particular printing devices.

The clear separation in architecture between revisable and printable forms is notional, and does not always appear in practice, particularly in the commercial world. First generation product manufacturers tried to improve products while remaining compatible with what already existed, because transfer to a new formalism was difficult. The result is that a number of products exist on the market using formats which are not always strictly classifiable as revisable or printable. In general, these formats are printable formats including escape sequences, treated as a kind of comment line, allowing revisable properties to be handled. For example, commands have been included in these formats to allow partial reformatting of a document or renumbering of sections, in cases of minor modifications to text, a development which corresponds to a real demand from customers but makes the formalism difficult to classify. This is typically the case for the Bull DSA 101 format or IBM's DCA. In this article, which attempts to survey architectural concepts, formalisms have been chosen that are typical illustrations of an architecture or an abstraction. The two preceding formalisms, although

they are widely available on the market, will not therefore be considered here.

Second generation systems are essentially an improvement on their predecessors. They maintain the structure shown in figure 1, with the same markup process, but have been adapted to richer typographical possibilities provided by non-impact printing. However, most of these systems, implemented before 1983, do not use page description languages in their output and suffer from the difficulty of inserting graphics and pictures in the the text. Second generation systems have also evidenced the notion of logical structures, the hierarchical decomposition of a document, relationships between elements. They enriched the first generation with adequate processing of those structures: maths formulas, tables, references, indexes, etc. However they kept the main characteristic of their ancestors: the input file to the formatter is constructed directly by the user, who has to know the syntax of the formatter command language and has to wait the final layout to discover mistakes in the markup.

Today, document preparation systems can take advantage of the facilities provided by new man-machine interface technologies. In particular, they allow screen displays of character strings in different fonts or different alphabets, as well as graphics or mathematical formulas which can be interactively modified. Why should a user learn a language and waste time (and paper) *debugging* the markup of a document, when a document can be created and modified directly on the screen? It is no longer necessary to mark up document contents to produce output effects. In a *third generation system* the user can operate by direct manipulation on the the only visible representation that appears on the screen; no knowledge of a particular representation used internally by the system is required.

The first systems of this new generation were sometimes referred to as WYSIWIG systems (What You See is What You Get) which show on screen exactly what will later be printed on paper. These systems had a number of limitations, which are now well understood:

- Screen resolution is significantly less good than that on non-impact printers. There is useless loss of quality if the paper layout is as poor as the screen one.
- the paper form of the document provides no control over logical structure. It is difficult to perceive how a numbered list, a tree structure

of sections, sub-sections and paragraphs can be simply modified when the system knows only sequential data.

- decomposition into pages does not ease simple editing tasks. It is often easier to work on a continuous document, at least at the level of a logical element.

This class of system, which some have called *what you see is all you get*, is being abandoned for richer systems (e.g. [JQ84, QV86, Fur86a]), which give both control of the document structure (and commands associated with the creation and modification of that structure) and immediate feedback on screen, e.g. showing emphasized characters in italics, centering titles, etc. This screen display is not an exact image of what will appear on paper, but, even if it is only a good approximation to what will be produced on paper, the user has a sufficiently realistic vision of the definitive version, which is generally enough to avoid the need for marking up.

For a third generation system, the revisable representation of documents therefore takes on a new meaning: as users can no longer access to it, it no longer needed to make it readable by a human. The complexity of this representation has in fact to be adapted to the functionalities of the system and to use a syntax suitable for reading by a program. It will, for example, contain elements such as pointers, binary-coded values or regular expressions.

Printable representation properties are examined in section three, while second generation markup languages are the subject of section 4. Two new formalisms that have appeared over the last few years, intended for third generation systems, are considered in section 5.

3 Printable formats

The main objective of a printable representation is that it should be independent of the printing device. The representation must therefore provide a certain level of abstraction which is then interpreted to produce an actual print. Several paradigms are possible to define this abstraction. The three most important are examined below.

The oldest abstraction is that of a virtual peripheral. A set of commands is defined for a virtual printer. Then for each hardware device, a piece of software (a driver) translates virtual printer commands into one or more actual printer commands. This method restricts practical applications to a class of hardware with similar sets of commands, and is not really device-independent. More important still, the image type that can be produced is

limited by the way the command set has been designed. The `.dvi` file of TEX is an example of such an approach, yet it is is one of the most powerful and has proved to be portable on a wide range of devices.

The second abstraction can be called a "declarative paradigm". In this approach, specific constructions (data structures) are defined to structure the contents of a document and to capture its properties. The document appears as a tree structure of operators or constructors articulating data. For example, the page structure will be defined as a block, with a few attributes, comprising a list of lines, each of which takes the form of an array of characters. This approach is fairly flexible if base structures can be combined to build up more complex ones. The image type that can be produced is clearly limited to the basic types and operators allowed (characters, vectors, circles, etc.). Insofar as any geometrical figure or graphic can be produced using a sequence of vectors in a standardized coordinate system, it is possible in fact to define any image on the basis of a formalism of this type, but at the cost of having to declare thousands of vectors. Here again, printer software has to take account of such a representation, analyse it and translate it to drive the printers.

The two abstractions above were valid for impact printers, but the problem has changed fundamentally since the appearance of non-impact printers. For such hardware a page is merely a vast matrix of dots which can be coloured individually. At the lowest level, the concept of a character or graphic no longer exist. Unlike impact hardware whose technology set an upper limit to image production, non-impact hardware no longer limits the type of printable image. It is only the *software interface which limits what is possible.* A new constraint is placed on representations intended for such hardware: universality, i.e. the software interface must not restrict the functionality available to users with respect to the underlying hardware.

The most universal interface is not that of the technology itself. Representing each page as a matrix of pixels would be very costly and would not be universal as it depends on the resolution of the equipment. What is needed is actually a general interface, where the page is considered as a geometric plane, where the user may have its own coordinate system, express geometric transformations and draw shapes which can be described mathematically without any device background. Over the last few years a new model has been established, with a few variants: page description languages [WW82], [SR84], [Xer84a], [Ado85a], which seem to provide the best level of compromise between universality, power, simplicity, performance and device independence.

The following two sections consider two formats that are typical representatives of their format category: the CCITT T73 format [CCI86] which uses the declarative approach, and the best known page description language, PostScript.

3.1 Recommendation T73

T73 is the CCITT recommendation concerning document structures for telematic services. As a consequence of the coordination between the CCITT and the ISO, this recommendation corresponds to the ODA-ISO 8813 standard. Initially the T73 recommendation concerned only the layout, or physical, structure of documents. As the ISO took over this recommendation and added logical structure to it, the CCITT has in turn incorporated logical structure into its recommendation. There is no longer any significant difference between ODA and T73, which can be treated as synonymous and which handle both a revisable representation and a printable representation of documents. In the section concerned with ODA (section 5.2), the problem of logical description and links between the two representations is considered. This section is concerned with layout representation.

A T73 representation follows a tree structure made up of nodes each of which has a type, attributes and hierarchical constituents. The various constituents of this hierarchy as follows:

Root. The root of the hierarchy represents the document. It is an element of type *page set.*

Pages sets. A *page set* is a collection of pages with common properties, e.g. all the pages in a particular chapter that would have as page header the chapter's title. A page set will have as ancestor a page set (may be the root). It will contain other page sets or simply *pages.*

Pages. A *page* is a logical concept independent of the paper page actually used. It defines a rectangular area available for printing and a coordinate system used as a reference for positioning the contents of that page. Pages are hierarchically attached to a page set. Pages contain either *frames* and/or *blocks*, or *content portions* directly.

Frame. A frame defines a rectangle parallel to the edges of the page and a coordinate system. It is attached to either a page or another frame, but it cannot be the leaf of the hierarchy. It contains either other frames, or blocks.

Blocks. A block defines a rectangle parallel to the edges of the page and a coordinate system. It is hierarchically associated with a page or a

frame. A block contains zero or several content portions, which will be printed using the properties of the block.

Frames and blocks are positioned with respect to the immediately higher level in the hierarchy and translate the coordinate system to their origin. Later coordinates are expressed relatively to the frame or block origin. To move the content of a block or a frame on the page, it is hence sufficient to change the block or frame position.

The two notions of blocks and frames are similar but have different purpose and different attributes. A block is a real piece of information while frames mainly serve to template the layout process. When designing a layout prototype to set the positions of titles, margins, page headers and footers, the designer will construct frames, rectangles containing no information at this time. The author building up the document will essentially bring blocks and the layout process will place the blocks into the appropriate frames.

Contents Portions. The contents of a document (what has to appear on paper) is divided into *content portions*. Every portion has the presentation attributes which define its appearance and must refer to a *contents architecture*. The contents of a fragment is a sequence of values requiring interpretation: a byte string may be regarded as a character string or as a representation of pixels in a digitized image. It is the contents architecture which defines how sequences of values are to be coded, and contents architectures have to be standardized to allow systems to decode contents.

The *imaging process* of such a structure to a physical device has to traverse the hierarchy in depth-first order, and to consider the attributes of the objects it encounters. The principle is that the imaging process, as it moves down the hierarchy, collects the information necessary at the level of page sets and then for each page prints the different content portions, applying the positioning rules imposed by frames and blocks. This depth-first order may be modified for a particular object by its *imaging order* attribute.

Rectangular zones may have non-empty intersections. In this case transparency is shown by the *transparency* attribute. If two superimposed objects are opaque, the last one displayed, indicated by the *imaging order* attribute, is the nearest to the surface and therefore the only one fully visible. It is possible for an element to have a confidentiality attribute, with respect to the operator who launched the imaging process, and therefore it will not be printed. The imaging process is more directly concerned with *presentation*

attributes which specify, for example, alignment characters (justified, ragged, centered), the character fonts each of which is associated at this level with an integer identifier, the direction of writing, the character set used, and the spacing of lines.

The most common contents architecture is Character Architecture. This architecture defines three possible content formats according to whether the document is formatted or not, as we shall see in more detail in section 5. This section only considers the definition of "formatted text". This text is made up of alphanumerics characters and control functions. The most important control functions are:

- Line-feed, Carriage Return, Horizontal Tabulation, Backspace and a few others are the usual control characters;
- Partial Line Up and Partial Line Down for superscript and subscripts;
- Reverse or Restore Presentation Direction to invert or restore the direction of printing;
- Select Graphic Rendition followed by the appropriate parameters, which in particular allow the current font to be changed. The integer parameter following SGR may specify a font or a change to a variant of the current font (italics, underline).

3.2 Page description languages

A page representation in such a formalism is also called a procedural page description. The description of the page in this case is a genuine *program* coded in a specific programming language, the page description language. Unlike a declarative approach which describes what is to be obtained, a page description specifies the sequence of operations necessary to produce the results. Hence, the main characteristics of a page description language is that each page is described by instructions which are executed directly by the printer (or a specific driver [BQ85]). And this execution uses classical programming languages constructions, assignments, computation of expressions using stored variables, conditional instructions, loops, procedures, etc. While T73 is a *static* formalism, page description languages are essentially *dynamic*.

Two page description languages are available on the market today: the most famous presently is *PostScript* by Adobe which has been adopted by numerous printer manufacturers, and *InterPress* which is marketed by (Rank) Xerox and supported by a group of computer manufacturers. Both use the same evaluation principle, the same imaging model, both are post-

fixed languages using stacks (see [Rei86a] for a more detailed discussion of these two languages).

All the items appearing in the document are pushed in turn onto the stack, apart from operators which are immediately executed. These operators take their arguments from the current execution context, most often the stack. For example, an addition will add to the two values at the top of the stack and replace them by their sum. In many cases a variable will be retrieved from a dictionary, a set of keys each associated with a value. Some global variables are under control of the system, such as current trajectory, line width and the current font.

A number of executable primitives are available to describe the image to be produced. Those primitives operate in a *user coordinate system.* There is a default coordinate system which can be modified at will to the appropriate metrics (inches, meters, typographer point, etc.). Within its coordinate space, the program then constructs pictures in terms of ideal, mathematically described, lines and curves. A series of such lines and curves makes up a trajectory, a *path.* To deposit ink on the paper, specific operators called *marking operators* have to be used. There are three basic marking operators: *stroke* to draw the path outline, *fill* to fill the inside with some color, and *show* for alphabetic writing. It is up to the language interpretor to map the current path in user coordinate to the best approximating dot matrix onto the device, based on global variables for the line width, line pattern, etc. Since the path is expressed in metric coordinates, on any device it will have the same size, and only the interpretor will deal with the device system. This is why a picture is really device independent.

Hence there is no basic primitive to draw a rectangle, a triangle or any specific path. In order to draw a rectangle, the current trajectory is defined as a particular set of lines in a few instructions, and it is then drawn on paper using the stroke operator. To print several rectangles, the instructions are iterated. If a great number of rectangles are to be drawn, it is advantageous to group the instructions together in a procedure. Alphabetic characters are also defined as trajectories, and a font is simply a set of trajectories. As well as standard fonts, every document can therefore define its own fonts, simply by applying a geometric transformation to those predefined in the language.

A specific operator allows to eject the current page from the device, in order to replace it with a blank page, saving or not the current context. In general, the structure of a 'document' is as follows: it begins with a preamble which defines a set of procedures, and then the document is a

'main program' which is often very short, essentially made up of calls to the preceding routines, interspersed with page feeds. To optimize document size and printing time, most manufacturers provide the possibility of loading the printer with the most commonly used procedures, which reduces the length of the preamble. Readers should consult the reference manuals [Ado85a, Ado85b] for a description of all the basic operators.

Amongst the most interesting properties of a page description language, the following should be noted:

- the possibility of defining geometric shapes in an abstract and device-independent way;
- the ease of definition of higher level objects, thanks to the use of procedures in the language.
- The possibility of using these procedures in different contexts, with parameters dynamically evaluated at calling time. For example, the same procedure *square* will be used to draw a square, a rectangle or a losenge, filled with black or grey, at any position on the page. To do so, a procedure *square* must be defined, which draws a unit square at the origin of coordinates. To draw a square, one simply calls the procedure. To draw a losenge, coordinates are rotated, to trace a rectangle different units are used on x and on y (e.g. 1 inch and 2 inches), before the procedure *square* is called.

Beside universality and device-independence, other advantages of page description languages include:

Compactness. Procedural page description languages are often more compact than non-procedural descriptions of the same pages, as they can use the power of the language to express regularities. For example, to draw a grid, a line drawing procedure is repeatedly called, while in T73 all the straight lines in the grid have to be enumerated with their coordinates.

Power. In a page description language, it is possible to write emulation procedures for another printer protocol. It is possible, for example, to use a page description language to analyse and print a file in T73 format, like the reverse is clearly not possible.

Modularity. The use of procedures allows modular construction. Each user or organization can build up libraries defining commonly used objects. For example, a procedure can be built up in PostScript to print an organization's logo. That procedure can then be called whenever the logo is needed; the parameterization of the language allows the same

procedure to be used whatever the size of logo required and its position on the page.

Adaptability. In cases where device-independence is very difficult to obtain, a procedure can be defined, that behaves according to the resolution of the printer actually being used. The algorithm can then be included in a library, as described above.

4 Markup languages

The first generation of systems gave way to a second which enhanced the first but maintained its main feature: markup commands are interspersed in a document which guides page formatting. The advantage of this method is that the quality of the terminal on which the document is produced has no influence on the final result. It is possible to create high quality documents even with a limited capacity terminal. Moreover, an author can use a preferred or familiar text editor to create a source document.

These systems have had to adapt to the non-impact printers that have appeared on the market, allowing the production of documents with rich typographical facilities. A system such as TEX [Knu84] produces results that are often of high quality because layout algorithms [Pla81], [PK82], [KP81], have had to be improved to approximate to those used by publishing professionals. New commands have also appeared allowing advantage to be taken of this typography, even though few yet are easy to use and produce good results in French typography. As software engineering has undergone significant advance at the same time, formatting systems have also developed towards becoming more powerful languages (parameterized commands, macros, etc.).

But the major trend in the evolution of languages has been towards document structuring. In the early markup systems, markup commands indicated operations controlling the the formatting process. For example, the NROFF `.ce` command centres the following line. This kind of language has been called *procedural* (procedural markup languages), because the author directly controls the execution of the formatting process by means of instructions. In using such languages, the following disadvantages appeared:

- The logical structure of the document is not expressed in the markup. This makes automatic searching difficult. It is not, for example, possible to search by program for all titles in a document and then build up a contents table. Assuming that titles are marked by an indent

command and a switch to bold characters, it is not certain that a search for those commands will find titles alone. Moreover, if an author has to take over a document from some other author, he may find it difficult to identify its structure.

- The concept of style is non-existent. Changes in style, e.g. to move from the *research report* style in one column to the *scholarly journal* style in two columns for the same text requires revising practically all the commands throughout the text. Moreover, the author is not always a typesetter (in fact he very rarely is), and the commands incorporated within the text are often not those that would be used by a publishing professional.

A solution to these difficulties of procedural systems was provided by the introduction of *declarative languages.* In the declarative approach[1], the author declares the logical structure of his document and the styles he used to assist him. The system is told what to handle rather than how to handle it. The markup symbols used are no longer formatting instructions but structure markers. For example, the markup sign `@(section)` can be used to show the start of a section, or again `<begin(itemize)>` to indicate the start of a list and then `<item>` in front of each element in the list. The system itself provides the necessary indentation for the paragraphs forming the list and will add the appropriate number in front of each paragraph. Since the system has been parameterized by typesetting professionals, the documents produced are of professional quality.

Note however, that a well constructed procedural system allows macros or procedures to be built up making it more declarative, while allowing the use of a procedural approach for users that require it. This is the case, for example, of the various document preparation systems under UNIX[2] [KLO78], that provides several such macro packages. The first well-known fully declarative system is the Scribe system [Rei80b, Rei83], which itself drew on the PUB system [Tes72] (the interested reader should consult [FSS82] for a more complete comparison of these different systems). Later, the LaTeX system [Lam85], which has been developed with the concept of an *environment* drawn from Scribe, on top of the TeX system taking advantage of the considerable effort that went in to TeX for layout of formulas and equations.

[1] Clearly, the distinction between declarative and procedural affects the level both of revisable representations and that of printable representations.

[2] UNIX is a registered trademark.

The concept of a style can be expressed by markup indications and used heavily. Again in the LaTeX system, the style is declared at the start of the document, with a general choice of operations. The command:

```
\documentstyle[twocolumn]{article}
```

produces the style "article" (in particular, the title in bold, and a summary using a longer line) printed over two columns. Eliminating the option `[twocolumn]` returns to a normal style.

As a part of the development towards declarative languages, a project has been launched at the IBM Research Centre [Gol81] based on the GML system [Gol78] [IBM80]. With the aim of imposing a standard in this field, the ISO finally adopted as its international standard the system produced by this project, with a few modifications, giving it the name SGML [Gol83] and making it standard ISO/DIS 8879 [ISO86b].

4.1 Standard generalized markup language—SGML

SGML is a declarative language which incorporates an idea drawn from PUB and then Scribe, in which the formatter is regarded as a document compiler. The source program for this compiler is the document to format, the object code is the printable form. In the case of a page description language, this analogy is reflected in practice since the printer actually executes the object code.

The difference between SGML and its predecessors, which used an ambiguous and loose syntax is that the source document is expressed in a rigorous language and the compiler can make use of the general compiling techniques. Parsing of the source document leads to construction of a tree reflecting the document structure. Code generation constructs the object document expressing the semantics of the language, taking account of the attributes associated with the nodes of the tree structure.

In fact, in the markup language SGML, the code generation need not be that corresponding to formatting. We could consider markup intended to ease the automatic translation of a document, or for building indexes to be used in search strategies in a document database. The range of SGML covers more than simply formatting.

Marking up in SGML is based on tagged parentheses: every element that requires marking-up is enclosed between left and right brackets, and the parenthesis is accompanied by symbols. The left-hand bracket, or *start-tag* starts with the character <, followed by a label (a character string, and any attribute assignments there may be, and is terminated by the character >.

The corresponding right-hand closing parenthesis or *end-tag* is identified by the same tag label, but preceded by the character '/'.

A chapter would therefore be marked up as follows:

```
<chapter>
Once upon a time...
...
... they all lived happily ever after.
</chapter>
```

Total parenthesizing is cumbersome for a user, and the SGML syntax allows the omission of *end-tags* where they are redundant. For example, to define a list, the end marker can be left out for each element.

```
<list>
<item> this is the first item. No end marker.
<item> this is the second item.
   Only the end of list marker is required.
</list>
```

Although declarative markup systems show the beginnings of a certain structuring process, in many languages that structuring remains limited to the tags defined by system designers, typically limited to the concept of sections, subsections, paragraphs and enumerated lists. An author is constrained to use this limited set of constructors. He cannot define his own structures, far less the properties of their structures. This is made possible in SGML through the introduction of a *meta-language* within the marking up system itself.

SGML allows the definition of new tags, and such a definition is called an *element declaration.* An element declaration defines a new class of entities with its structure and properties. The name of the class is the label which will be used subsequently to mark the presence of the element. An SGML system can therefore be adapted to the structure of documents appropriate to the operating environment. The value of the meta-language is that it can appear within the document itself, the syntax being that any marker beginning with the character[3] '!' in fact introduces a meta-language element. For example, the following construction could be used:

```
<!ENTITY % title "#CDATA" >
<!ELEMENT
01 article      (%title, abstract, section+, references?)
version (draft \| final)
```

[3]Special characters can be redefined at will.

```
02 abstract      p
03 section       (%title, (p \| list \| fig )* )
04  list         (p+)
   type    (simple, enumeration)
05 fig          (graphics, caption)
   xsize, ysize  #NUMBERS
06 graphics   #CDATA
07 caption       p
```

The first construction used, introduced by the symbol `ENTITY` makes it possible to define an entity which will be reused subsequently, a kind of 'macro'. Here, a title is simply defined as a character string.

The second construction defines the class of documents article. It shows that an article is formed of a title (a character string) a summary, sections (`+` indicates that there is at least one, followed optionally by a bibliography (the option is indicated by the `?`). An article has the attribute `version` which takes the value draft or final.

A section, in turn, also has a title, followed by a series of n (the symbol * indicates that $n \geq 0$) objects which are either paragraphs, lists or figures. A list is a sequence of paragraphs, and it can be a simple list or an enumeration. In the latter case, each element in the list has to be preceded by its rank 1),2), etc.. A figure includes a drawing and a caption. The attributes `xsize` and `ysize` indicate the space allocated to the figure.

Initially, four basic types are defined by the lowest level constructions: character (CDATA), elementary paragraph (p), numeric values (NUMBER) and the IDREF type which is used for an SGML object identifier. This IDREF type makes it possible to build up cross-references within a document. The meta-language is reasonably complete and allows a wide variety of structure descriptions. The interested reader should consult the text defining the standard [ISO86b] for more precise and exhaustive information on the syntax of SGML.

The fact that new elements can be described introduces a problem: a system receiving a description of structures in the meta-language knows their syntax but does not know their semantics. Although the system can detect that a document conforms to structure declarations and can test the validity of attributes, it cannot discover how to format these new objects, or how to use their attributes. An elementary method is obviously to add these semantics to the system by including procedures defining semantics. But then documents will no longer be portable into another environment. The solution provided by SGML is the possibility of linking new elements

with those already existing in a system. For example, to tell the system how to format an abstract one writes:

```
<!LINK abstract p indent=10>
```

The LINK marker shows that an abstract must be formatted as a paragraph but using an indentation of 10 units. Note that this link mechanism is, however, less powerful than a parameterized macro mechanism in a system such as TEX.

In a document, it is sometimes necessary to call on external files. This introduces a link with the local operating system, and two solutions are possible a priori. The first, used for example in LATEX, is to introduce a specific instruction in the language to reference files. The author writes, for example, `\include{filename}`, and it is up to the local implementation of the LATEX system to manage file access. If the document is ported onto another operating system, these commands may not work properly.

The SGML solution is more general but more complex. In fact, a complete sequence of commands for a particular operating system can be inserted within a document. As this document is not portable, an additional flagging system allows a given system to consider only commands which concern it. A magic formula such as

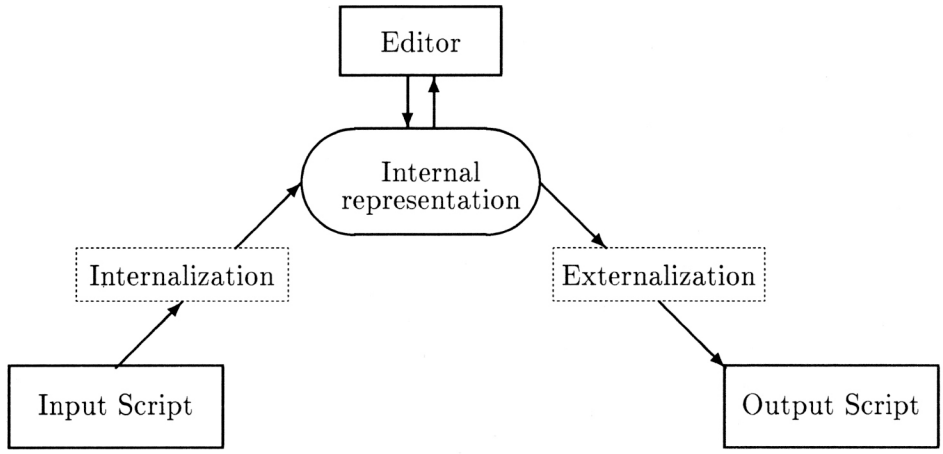
```
<![BOZOS;[<? BOZOS commands>]]>
```

allows commands in the BOZOS environment to be executed.

5 Third generation representations

It should be remembered that the characteristic feature of these representations is that they are in principle handled only by software. They are difficult or impossible to read for a human operator and are far more complex than the formalisms in the preceding generation. This section discusses two of these formalisms: ISO standard 8813, better known by the acronym ODA, and the Interscript formalism. The latter, developed at the Xerox Research Centre [AHLM84], has not enjoyed the success for which its creators had hoped. No commercial system uses it today, probably because it introduced too many new concepts to be adopted by manufacturers, and because it could not provide direct compatibility with existing standards. Some of these new ideas have, however, been incorporated into ODA, or have been taken up by university research projects.

Figure 2: Processing principle in Interscript



5.1 Interscript

In third generation systems, the internal representation of a document is built up in central memory by an editor, while the visual representation is displayed on a high-resolution screen. Another representation is *externalized* on a magnetic device when the user completes a session. Interscript calls this externalized representation a *script.* In subsequent sessions, the externalized script is again reloaded into central memory, i.e. it is *internalized.* The internalization process (figure 2) goes from a public representation, on the magnetic medium, to a private representation in memory, and vice versa for the externalization process.

The fundamental principle of Interscript is that it should be possible to exchange documents between heterogeneous systems with different capabilities but which do not need to know the structures used by the others. Suppose, for example, that one software package handles complex graphics (a package of the MacPaint type), while a spreadsheet allows tables of values to be built up (as in Multiplan). To build up a document containing both tables and figures, one might have to work on the same document in both systems. For this to be possible, each of the systems must be able to exchange a document with the other, i.e. to internalize a document constructed using the other package. Then it may update its own data, it does not need to be able to make changes to the fragments constructed by the other system, and clearly it must not affect the integrity of those data. The problem is that it should be able to read from a document the appropriate data.

The approach used by Interscript is similar to that used in software engineering: general purpose programming languages are used to solve the problem posed, in which specific data structures and algorithms are described. Interscript sets out in the same way to define a basic language for the description of documents, used to express their particular structures, with the aim of ensuring the language can describe the whole range of structures, from the properties of a character font to the logical or layout structure of a document.

This basic language is then used to describe higher-level constructions. To exchange documents, heterogeneous systems do not need to know the semantics of these constructions: the syntax of the basic language is sufficient. When a system receives a script, which only depends on the basic language, it can internalize it. Its capacity may stop there. If the system recognizes none of the constructions used in the document, it will be able to do nothing with them. But if, for example, it can handle the concept of a *table* it will be able to handle all the *tables* in the document, and only the tables. For the same document, another package will handle the other constructs.

Therefore, there are clearly two levels in Interscript, the basic language and the use made of it to describe documents concepts, which are either data structures such as paragraphs, character fonts, or operators which lead to a particular type of processing. The existence of the basic language makes it possible to avoid specific content architectures, as will be seen in ODA, and therefore ensures that the system is open. If it becomes necessary to define a new contents architecture, the organization of information for a certain class of objects, it will be defined in terms of the base language, without having to modify or extend the standard. At this level, the approach is similar to SGML, the main difference being that expressions in the Interscript language are evaluated dynamically, while all marking in SGML is a static description of the structure and its attributes.

The disadvantage of this method is that compatibility with existing systems is not immediate. To recover existing documents, they have to be transformed into the syntax of the Interscript base language. Although that conversion does not seem to be particularly complex, this is undoubtedly one of the reasons why Interscript was rejected as a standard [JSP85]. ODA, on the other hand, sets out to provide a 'bridge' over the other standards, at the price of sacrificing more advanced technology. It is supposed to be easier to convert existing documents to the ODA standard; this compromise representing an industrial constraint.

Another possible consequence of openness to various structures is the risk that each is unaware of the structure of the other. This is unsatisfactory in a standard. In order for systems to be able to exchange a minimum of information, Interscript also proposes a set of so called "standard tags" constructions and operators which it recommends rather than a construct with equivalent semantics but not recognized by other systems. The basic language was used to define the usual concepts of font, paragraph, title, list, tables and others, and also to define operators expressing the formatting operations, which have proved to be very valuable.

5.2 Basic language

A document coded in the Interscript representation, a script, has to be internalized to be used by an application. Internalization is an evaluation mechanism which constructs the internal representation of the document (cf. figure 2) The key concept in Interscript is that the internalization stage is not a simple translation of tags to pointers but a dynamic evaluation. Here again the approach is procedural: the script stored on disk is in fact a *program* whose execution leads to the construction of structures in memory. Similarly, externalization is a relatively complex process which constructs an expression that can be evaluated. The internalization of a script requires the use of a stack, storage of intermediate variables, many operations usually associated with the interpretation of a language. The basic Interscript language has a simple syntax (20 or so rules), whose semantics guides the interpretation of the script.

A script consists of a tree structure, in which each node is labelled with tags. The general principle of the language is that each node in this hierarchy has a content whose interpretation can only take place by consulting tags. The lowest level of this hierarchy contains bytes which can be interpreted as integers, characters or the pixels of a picture.

Tags provide the information necessary to decode the contents of a node in the tree. Moreover, as in an object-oriented formalism, the attributes expressed at one level of the hierarchy are inherited by all subtrees qualified by that tag. A document therefore takes the form of a tree whose root has the tag DOCUMENT. The syntax to denote a subtree is to enclose it between curly brackets. A document therefore takes the following form

```
INTERSCRIPT{DOCUMENT$ contents of the document}ENDSCRIPT
```

A tag is different from a type; when we say that a document is of type X, this implies in general that it cannot be of any other type. On the other

Figure 3: Part of a script.

```
{DOCUMENT$
   -- start of document
 figureCounter == 1 -- global variables for the document
 leftMargin == 254
 {SECTION$
   SECTION
     -- paragraphs appear here
    {FIGURE$
     {IMAGE$
        IMAGE.coding = PostScript
        {POSTSCRIPT$
        -- a PostScript program describing the figure
        }
     {CAPTION$
         CAPTION.counter = figureCounter| + 1
     PARAGRAPH$
        PARAGRAPH.leftMargin = leftMargin| + 127
        PARAGRAPH.font = SECTION.font|
        {CHARS <this is a script example>}
    }}
   }
   -- rest of document
   }
}
```

hand, a node in a script can be labelled with several tags each of which provides different information on the contents of the node. This is somewhat like multiple inheritance in object oriented languages. The authors draw a comparison between a node and a bottle: several labels can be stuck to a bottle, each of which provides a different item of information, such as a price or a name, without changing the contents of the bottle. The semantics of these labels is not necessarily understood by all the systems that will handle the document. For example (figure 3), the caption to a figure will have the two tags **PARAGRAPH** and **CAPTION**. The **PARAGRAPH** tag shows that the content of the caption is to be regarded as a paragraph

which will have to be broken down into lines if necessary and enclosed between two margins that are determined by the attributes of this paragraph.

The fact that this paragraph is also a figure caption which has to bear a label *figure i* (where *i* is a counter to be updated on the basis of the number of figures), depends only on the `CAPTION` tag. The contents of the caption could be a graphic instead of a paragraph: the two are independent.

Unlike ODA, the attributes of a label are not static but are evaluated as a function of the context, which makes it possible to obtain values dynamically, and above all to maintain consistency between values automatically. In the following example, the attribute `leftMargin` in the caption is evaluated to the current value of the margin in the section including the figure. If the figure is moved to another section or another document, the positioning of the margin to the figure will remain valid. The advantage of this dynamic approach is that it allows the use of contexts in this way.

The basic language includes a certain number of facilities which are not described here (see the review texts [AHLM84, JSP85] for more complete information).

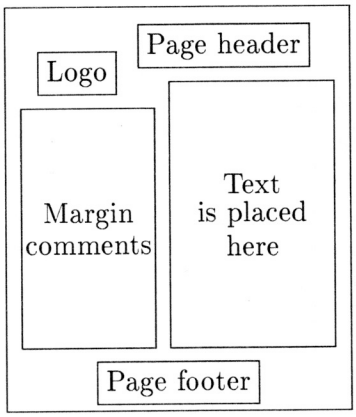
5.3 Formatting

The users of a document preparation system are not (in general) interested in the microscopic aspect of page layout, i.e. after which word a line break or page break appears. On the contrary, they expect the system to deal with these problems for them. On the other hand, they are interested in the macroscopic aspects: where the main heading appears, where the logo is placed if there is one, where figures appear, etc.

InterScript provides a page layout description mechanism to this end. This description does not exclude the concept of a style, its aim is to ease the task of a style designer, and also to describe the page layout of each type of construction (cf. the syntax-semantics problem in SGML). The page layout description naturally takes place on the basis of constructions and operators defined in the base language. These operators, placed within the document, will template page layout.

Page layout is formalized on the basis of the following analogy: page layout is the process of *pouring* a certain *fluid*, the contents of a document, into *solid containers*, essentially rectangles. These rectangles are not totally rigid, they are elastic containers that can take contents within certain limits, like boxes in TEX [Knu79a]. The way in which the fluid flows into the

Figure 4: Layout describing the appearance of a page.



container is specified by a *template*. The typical structure of a flow is as follows:

```
{POUR$
  POUR.template = {TEMPLATE$ -- flow template -- }
  -- the contents of this node is the fluid
}
```

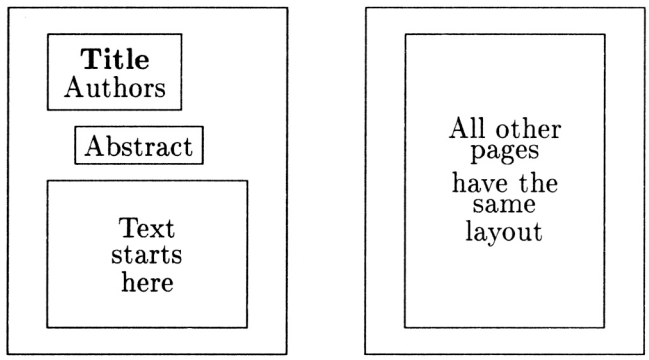
The template is essentially a hierarchy of boxes, also described in the basic language, which have various properties. Consider the following example, of a page (cf. figure 4) for it is desired to print a header (at the top) a logo and text leaving space for marginal comments.

The template for this page will show that the container is indeed a page and the page contains three boxes specifying the respective positions of the elements. In particular, the box containing the logo must directly contain all the information necessary for the logo, either a graphic in a page description language, or characters.

When flowing takes place, the fluid must not be poured into this box. Text and marginal comments must also only be poured into the boxes respectively allocated to them.

The system ensures that information is correctly distributed by using a *mold* mechanism (`MOLD` tag). When the pouring process constructs the page above, it directly places boxes with no `MOLD` tag on the page, i.e. the header and the logo. When it meets a mold node, it will find that the node has a label and it must therefore pour in the contents of the node with the same label. A template gives very important information on whether its use is *sequential*, *iterative* or *alternative*.

Figure 5: Layout describing the first page and the others



For a sequential template, the pouring process attempts to fill the boxes in the template one after the other. If the boxes in the template are full before all the contents are used up, the remaining contents are not printed, and this is an error. In an iterative template, if the boxes are full and there are still contents to pour, the template is restarted, and so on until all the contents are used up. As the concept of a template is recursive, it is possible to place other templates with different properties within a template.

To define the model of a document whose first page has a different format from all those that follow, as shown in figure 5, a template similar to figure 6 could be used, which itself includes a sequential template for the first page, and then an iterative template for the others.

5.4 Office Document Architecture

As pointed out earlier, in the section on printable formats, the ODA standard [ISO86a] is both a revisable and a printable format. This reflects a desire for optimization: in a traditional system, most of the document is generally its contents in the form of characters, and these characters are duplicated in the revisable and printable formats. If both forms are stored on disk at the same time, the degree of redundancy produced can become critical; the present document, for example, contains around 50,000 characters. To avoid this duplication, ODA aims at representing both the logical and layout structures simultaneously with no redundancy of contents. In this case, the procedural approach is difficult and it becomes convenient to adopt a declarative formalism with two tree structures pointing to the same leaves of contents.

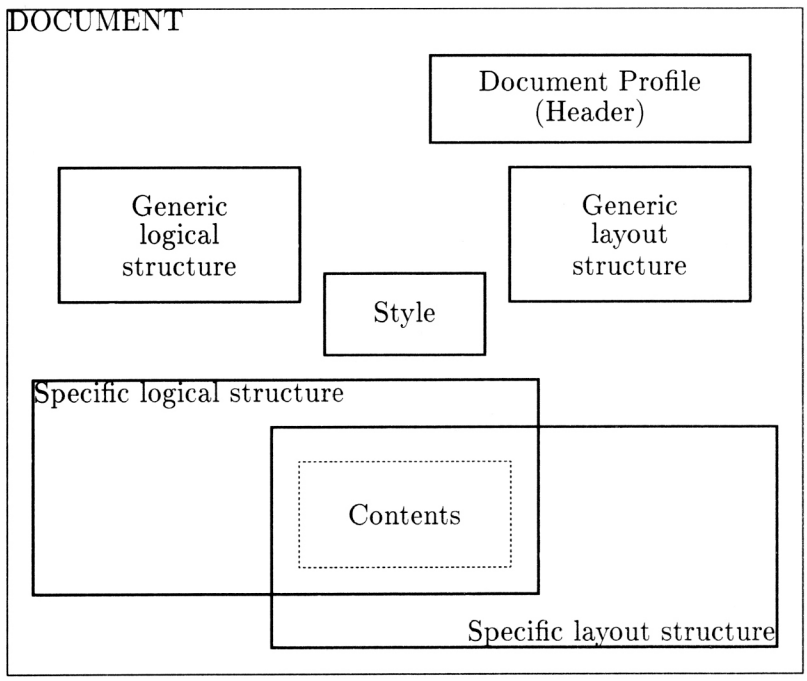
Figure 6: Script for multiple pages layout

```
{POUR$
  POUR.template =
        {TEMPLATE$   TEMPLATE.expresses = sequential
            {TEMPLATE$   TEMPLATE.expresses = sequential
              -- template for first page
            }
            {TEMPLATE$   TEMPLATE.expresses = iterative
              -- template for all other pages
            }
        }
  -- contents to be poured
}
```

Another reason quoted for the choice of a double structure is performance on small systems. In a system with separate structures, for example TEX, the author works on a revisable document, then generates the printable format, but this operation is a long one on current microprocessors (a second per page on a type processor is fast). For small modifications, e.g. corrections to spelling errors, not leading to page layout modification, some wordprocessor manufacturers prefer direct modifications to the printable structure in order not to reformat the whole document. This choice of a double structure, although it provides improvements in terms of load, does lead to new problems:

- There is only one version of the document. During editing of the document, it is often useful to have the old printable version, while modifying the new revisable version. If ODA users for this reason copy ODA files to preserve the old version, the result will be the same as for a traditional system, or even worse.
- The double structure introduces additional complexity. The most obvious is the problem of line and page breaks. The revisable form in principle contains only significant text; the formatted text has marks inserted by the formatter to indicate line endings. In order to update a document it will be necessary to start by removing these line endings. So that a software can know what it is dealing with, ODA has been

Figure 7: Organization of ODA documents



made to distinguish three forms of contents, the *processable form*, the *formatted form* (the only form considered in section 3.1), and the *formatted processable form.*

This complexity will be propagated in implementations, and there is even a risk that certain implementors will produce systems which process only one form. Users will once more face the problem of managing not only several forms but also be required to use several software packages to handle incompatible forms, even when all of them are "standard".

The overall organization of an ODA document is as shown in figure 7. Before commenting this structure in detail, it is necessary to explain the concepts of *specific structure* and *generic structure.* Consider the example of two reports $r1$ and $r2$. Assume $r1$ includes the title, an abstract, three sections and a conclusion. This constitutes the *specific structure* of $r1$. Report $r2$ also has a title and an abstract, but it has 4 sections before the conclusion. The specific structures of these two reports are different, but they are based on the same model, as are all the other "reports": they all have a title, an abstract, a variable number of sections

and a conclusion. This means that classes of documents can be identified which are all built up on a typical model. This model identifies the generic structure. These structures are the logical structures of the document, thus they are called *generic logical structure* and *specific logical structure.*

In an ODA document, there are both generic and specific *logical* structures, and also generic and specific *layout* structures. The specific layout structure is the structure consisting of pages, frames and blocks examined in section 3.1. The generic layout structure allows a formatter to build up the layout structure of the document by getting the information about margins, title positions, etc., in the generic layout structure.

All these structures are hierarchical. But rather than using a parenthesized syntax, ODA has opted for notation by reference: every constituent of a structure has a unique identifier and all the objects which are not leaves of the tree structure have a subordinates attribute. This attribute gives the sequence of unique identifiers of its hierarchical inferiors. If, for example, the abstract of a report is identified by 123, the first element of the subordinates field of the report element will be number 123. This reference mechanism makes it easier to construct complex documents some of whose parts refer to others.

5.5 Construction of a document

During the initial construction of a document, the editing process has to use the generic structure to construct the specific structure. To do so, it should use a particular attribute of generic structures called a *constructor for subordinates.* The value of this attribute is an expression formed of constructors or identifiers. There are three constructors called AGGREGATE, CHOICE and SEQUENCE. Each constructor handles a number of *factors* which are marked as optional, mandatory or repetitive. A *factor* is in turn either the identifier of another generic object or a new construction. For a report, we might for example have an AGGREGATE formed of a *body* and an optional *index.* There is no ordering relation within an aggregate, so the index to a report, if there is one, can be either at the beginning or at the end. The body of the report will be defined as a sequence containing the following in the order shown: an mandatory title, an mandatory abstract, a repetition of optional sections, and finally an optional bibliography, itself formed of a sequence of references.

Using this information, the editing process can create the specific logical structure of the report, interactively asking for optional information or rep-

etitions. When a component is created, (e.g. a section) the process executes the operation recursively, consulting the generic logical structure of the section. At the lowest level of the generic structure, the user indicates the possible contents of these elements. If, for example, we define the element paragraph we indicate that this element cannot have subordinates within the hierarchy and that the only contents portions allowable for this element are of the character type.

Allowable contents are currently being standardized, for the moment, three types of contents are clearly defined: character contents, already defined in the standard, geometric graphics contents which essentially use ISO standard 86332 (Computer Graphics Metafile) and photographic contents.

6 Page layout of the document

When a document is complete, and its specific logical structure has been entirely constructed, the formatting or *layout process* can be launched. This process uses the specific logical structure like an Interscript fluid and the generic layout structure as a container to construct the specific layout structure in pages and blocks.

In order to template this generation process, there must be links between the logical structure and the generic layout structure, essentially made by means of *layout references* and *layout categories.*

The layout object class attribute associated with an element of the specific logical structure indicates that the whole logical subtree rooted on this object has to be laid out in pages using only one instance of the generic layout class in question. For example, for a chapter of a document *page set* will be specified such that the pages of the chapter will be associated with this *page set*, and each even page of the chapter will have its title as its header.

In order to create a specific layout structure (pointing, it should be remembered to the same contents pages as the specific logical structure), the layout process uses the generator for subordinates attributes as in the creation of the specific logical structure. If a generic layout object specifies a generator for subordinates with an iterative construction, this component will be iterated as often as necessary. A *page set*, for example, specifies an iterative sequence for a page. This page is clearly a generic page, is used as a page model and in turn contains generic frames.

Moreover, in order to find out what type of logical object has been placed in what type of frame, it is possible to use the concept of a *layout category* which is simply a name. This name can appear as a layout category attribute

of a logical object and it can also appear in the permitted category attribute of a frame. If, for example, a frame model with the name *paragraph* is found in the generic layout structure, all logical objects referring to *paragraph* will be formatted with the attributes of this frame, (e.g. its line length). It can be seen that the *layout category* mechanism is similar to the concept of a name in InterScript, in that it shows the layout of a logical object must belong to a class of generic layout objects. In fact, after InterScript was examined and rejected by an ISO working party, some of its concepts were incorporated to as enhancements to after being transposed to a declarative universe.

The attribute fill order allows frames or blocks to be placed in a particular way. By default, blocks are placed internally within a frame from the top of the page to the bottom, and in the opposite order if the fill order attribute is set. By placing two blocks in normal and reverse order, it is possible to place the text referring to a footnote and the note itself. If necessary, several notes can be placed starting from the bottom of a page and working up.

7 Conclusion

We have described second generation systems as the document preparation systems currently being used. Several systems in this generation have won themselves a good reputation and a large number of users, in particular TROFF, Scribe, [S]GML, TEX and its cousin LATEX which was used to produce this document.

The ISO decided to standardize on one of the multiplicity of representations available and picked up SGML. The advantage of SGML is that it provides a syntax (and only a syntax, claim its critics) for describing document objects. Using this well defined syntax, it is possible to conceive of translators for a specific language into SGML and conversely. This approach at least allows document interchange between heterogeneous systems and the recovery of existing documents, even if a number of translation errors require manual correction. As the standard states, SGML (and other second generation systems) offers the millions of users of traditional terminals the possibility of producing high quality documents.

At the same time, the ISO has moved into unchartered waters by defining a standard corresponding to third generation systems before any has reached the market, hoping to avoid the incompatibility problems of the preceding generation which would not have arisen had manufacturers all adopted the standard. It is clearly difficult to define a standard for the future before

practice has allowed the concept to be validated; moreover, participants in the working parties of the standardization organization are not in general involved in research or development work which is at the same time defining the new systems. In addition, for an industrial standard, compatibility with existing products has to be maintained.

The ODA standard was clearly designed for third generation systems, extrapolating concepts introduced by the second, notably the concept of the logical structure of documents. By deciding to maintain a double logical and layout structure within a single document, ODA has had to adopt a declarative view of its representations, and could not follow the new technological avenue opened up by the procedural approach characterizing page description languages and more recent experimental work. In addition, a distinction has to be made within ODA between the concept of logical structure and layout structure.

It is not reasonable to look at T73 as competing with PostScript or any page description language. When arriving on the market, the ODA systems will very likely have only page description language based printers, as, one after the other, the manufacturers of printer hardware are all adopting interfaces of this type. As Brian Reid points out [Rei86b] the use of page description languages is limited more by the imagination of users than by technology. It would be surprising to see ODA layout structure printers, yet this will surely be emulated by some PostScript printers.

A page description language is totally satisfactory as a printer interface, and this indeed is its aim. But for other purposes than printing, the declarative approach of another format, such as T73 may have advantages. For example, one can imagine a bottom of the range word processing system working directly on a printable format, while interactively modifying a page description in PostScript is difficult (yet not impossible). On the other hand, it is easy to produce an interactive editor which acts directly on the declarative ODA layout format, as long as it is taken into account that the logical structuring aspect of a document is missing, and then translate the representation into a page description language in order to print it. This non-academic approach may be of interest to manufacturers, simply because it is profitable.

Since T73 is a CCITT standard it is likely to be used for the PTT fax services. The problem here is not simply to print but also to recognize. If the image analysis device in a fax machine can see rectangles containing information within a page, then recognize the characters within these rectangles, we reach a structure of information close to T73, which becomes a

good target, with great advantages in terms of transmission cost with respect to the pixels used today. It is clear that these two types of formats are not really concerned with the same commercial markets and that a manufacturer could offer both a page description language and T73 in its catalogue without inconsistency.

At the level of the logical structure of documents, ODA represents a considerable progress over existing word processing systems. The ODA standard has taken over the best concepts inherited from the second generation: logical tree structure of documents, attributes of components inheriting generic properties, and the concept of style for formatting. The absence of a basic procedural language, and considerations of compatibility with existing systems has led to a concept of contents architecture which now constrains the ISO to standardize all the possible contents of a document. In general, the standards from other fields will be simply taken over with an architecture for character contents, an architecture for graphics contents, for geometric contents, for pictures, etc. The method is cumbersome, the system's openness towards new contents is limited a priori, but this provides the upward compatibility required by many customers. It is however regrettable that ODA has not defined an architecture incorporating the features implemented in the systems producing high quality typesetting for documents containing chemical or mathematical formulas.

Commercial third generation systems will surely have the possibility to produce ODA documents. However, ODA has been designed explicitly as an interchange format, not to be accepted readily by editors (a major difference with Interscript). Thus, it is likely that many new systems will use internally some ad-hoc representation, may be a standard one like SGML, (particularly in environments already using its predecessor GML), and convert into ODA at the very last moment for interchange purpose.

It is certain that only research systems will use basic languages and a procedural approach for the description of documents as in InterScript, but this may allow the preparation of future standards.

Electronic Mail of Structured Documents: Representation, transmission, and archiving

Brian K. Reid

DEC Western Research Laboratory and *Stanford University*
100 Hamilton Avenue
Palo Alto, California 94301
U.S.A.

1 Introduction

Considerable attention has been given recently to the issues of representing, modeling, and manipulating structured documents. Regardless of the representation and regardless of the manipulation technique, the usual reason for making any document, structured or not, is to show it to another person. There is usually a separation in time or space between the creator of the document and the user of the document. The difficulty in communicating and storing structured documents comes from the need to make sure that the reader will interpret the structure in the way that the writer intended.

One way of thinking about a structured document is that it is a document employing "late binding." In other words, many of the decisions about the form or content of the document have not been made yet. Ultimately, when a document is printed on paper or shown on a CRT screen, all of its structure is removed, by making all of the binding decisions that were left unmade in the structured form.

When communicating a structured document to another place or storing it for another time, the difficult part is to make sure that there is enough information included with the document that equivalent bindings can be made by the reader, to produce a printed or displayed unstructered document that properly resembles what the writer had intended.

The problem of communicating structured documents is fundamentally unsolvable; it is almost unsolvable by definition. This is because the only way to be certain that the structure will not be misinterpreted, that the binding decisions will be made properly, is to make all of the binding decisions and remove the structure from the document. This reduces the problem to one that is quite solvable, namely the communication of an unstructured document, but also that is not nearly as useful.

2 Structured and unstructured documents

The concept of "structured document" means different things to different people. I will not try to define the term, but I will offer several observations about systems employing structured documents:

- Structured documents contain more information than just the text or graphics itself. That information is usually called "the structure."
- Most structured documents must be processed or compiled or formatted into some concrete form before they can be printed or displayed.
- The compilation process discards information: the same structured document can be processed into several different concrete documents.

Even as there can be structured and unstructured programs that perform the same computation, there can be structured and unstructured documents that have the same content. The structured document is preferred for the same reason that the structured program is preferred: it is much easier to modify and maintain a structured document, and it is generally easier to adapt it to a new context. In the case of the structured program, the translation from the structured source-code form to the unstructured but executable machine-code form is performed by a compiler. In the case of a structured document, the translation from abstract to concrete, structured to unstructured, is performed by a formatting program.

Structured documents generally take advantage of what is called "late binding," namely postponing the decisions as long as possible. Specifically, when a structured document is stored in a file, there are unmade decisions represented in it, and the document cannot be printed or displayed until those decisions have been made. In the process of making those decisions, it is necessary to use information outside the structured document. The collection of information that is used to bind the unbound decisions is called *the environment or the execution environment* for the structured document. As a simple example, for a text formatting program, the image resolution in dots per centimeter, the size of the paper, and the set of available fonts are all part of the environment might include some indication of whether or not the imager can image in color, whether the inks are transparent or opaque, and so forth.

When a structured program has been compiled into machine code, the process is not considered to be reversible, because there is information lost in the compilation process. The names of variables, the type declarations,

the block structure, the physical layout of the source text: all of these vanish in translating the program into machine code. When a structured document has been formatted into a concrete document, that process is also not reversible. The structure is lost, for precisely the same reasons that it is lost during program compilation.

3 Storage and communication

The difficulties encountered in transmitting a document, in moving it from one place to another, are the same whether the document is moved in space or in time: if a message is transmitted to another location, the receiving environment is likely to be different from the sending environment in ways that are not obvious to the sender. Similarly, if a message is stored in an archive for some time, and then retrieved into the same location, the environment is likely to have evolved in time in ways that were not obvious to the originator of the message. In both cases, communication can take place if the sender anticipated the needs of the receiver or if the receiver is able to work intelligently at converting the document into a usable form. If the sender is not willing to anticipate and the receiver is not willing to think, then the only way that communication can take place is for the environments to be identical.

When a program is distributed to another site, or retrieved from having spent a long time in a library or archive, we normally speak of having to *install* the program. This means that a programmer must study the program and modify it so that is will compile and execute properly in the new environment. It is quite rare for a program of any complexity to work upon immediate arrival at a new place. By contrast, when a computer mail message is distributed to another site, or retrieved from an archive, we expect to be able to read it immediately, without devoting much thought or spending much time on any kind of installation or conversion process. Yet, as we have pointed out above, the structure of a structured message can be every bit as rich as the structure of a structured program. Yet people expect the message to "work" at the receiving site without installation and without attention by a programmer.

To install and process and print structured documents when they are received, one of the following techniques must be used:

- The receiving environment must be an exact copy of the sending environment, so that all of the binding decisions will be made identically at the receiving end.

- The receiving environment must be specified and implemented precisely to some universal standard which is known to the sender, so that the sender can anticipate all of the decisions that will be made.
- The sender must anticipate every possible variation on the environment at the receiving end, and define the structure of the message in such a way that adaptation will be automatic.
- Judgment and intelligence must be exercised at the receiving end, to adapt the structured message to an environment that was perhaps not forseen by the sender.

The communication of structured objects is not unlike the communication of any other complex entity or idea: if the receiver is smart enough, the sender does not have to work very hard; if the receiver is unknown, then the sender must anticipate and provide for a huge variation in the nature of the receiver. The penalty for failing to do this is the failure to communicate.

4 Communicating by removing structure

Today the standard technique for communicating structured documents is to communicate the least common denominator: remove the structure, turn the document into a character stream using newline and newpage as its only format effectors, and then transmit that text file over the communication link.

By making all of the decisions before transmitting the document, the sender can ensure that they will not be made incorrectly. However, making all of the decisions before transmission reduces the usefulness of the transmitted document to the recipient.

Even this technique works only because there are certain minimum standards for character devices: almost all display and imaging devices are at least 80 characters wide, and almost all of them implement at least the ISO646 character set. Without these minimum standards it would be nearly impossible even to communicate unstructured documents.

5 Preserving lexical structure in communication

Whatever might be the structure of the document when it is originated, if that structure is damaged in the communications or storage process, then the chances of success in receiving and processing the document are reduced. All modern communication protocols are free of errors introduced by the communication medium; the difficulty is not that an "a" might be replaced

by a "b" in transit. The difficulty is that unless the sender and receiver have very similar environments, the representation of the structure might undergo mutation in transit. For example, if the sending computer has 36-bits words, and if the sending computer uses "word" as a data type in structured documents, then there will be problems.

The problem of preserving syntactic structure in communication has received a great deal of attention recently from international standards committees; the CCITT X.400 series of standards all address the issue. My own experience is that X.400 is a wildly excessive solution to the problem, and that a character-string representation such as a simple LISP S-expression is more than adequate. The details of this representation issue are not relevant to the scope of this paper and I shall not discuss them further: we shall assume that it is possible for the sender to transmit enough information that the receiver's data structures will be what the sender intended.

The true problem is preserving semantic structure in communication: if the receiver's data structure is intact but unintelligible, then proper communication has not taken place.

6 Semantic structure in communication

Using the model of Scribe or SGML to speak of a structured document in terms of *tags*, the *semantic structure* of a document is the meaning of the tags. The problem of communicating a structured document really reduces to the problem of preserving semantic structure across the communications boundary; all of the other problems are simple and will yield to straightforward engineering solutions.

As is always the case with semantic issues, the superficial solution is very easy and the full solution is very hard. For example, consider the case of communicating a document that contains a program listing, of some piece of source code in some programming language. The creator of the document probably set up the document structure so that the line breaks will occur where he intended: some variation on what is often called "nofill" mode. Under normal circumstances, if the receiving environment guarantees to put the line breaks in the same places that the sender has asked for them, then the document will be received properly. But if the receiver's environment is extremely different from what the sender intended-perhaps it has a very narrow display, or fonts with unusual character widths, it might not be possible for the receiver to put the line breaks where the sender intended. the receiver's response can be to truncate the lines, to shrink or rotate them,

or to abort the formating and demand human intervention. When a human intervenes, he will study the source code to determine a new set of rules for locating line breaks, and then either do the line breaks himself or run the source code through an appropriate prettyprinter.

Had the sender anticipated this situation, he could have encoded different structure in the structured document. For example, instead of using a "nofill" tag he could have used a "Programming language" tag or a "Pascal" tag or a "UCSD Pascal release 3" tag. As the tags become more specific, the possibility of automatic recovery from errors becomes greater, but the probability of the receiver's having a definition for a very specific tag becomes lower.

This communication situation is reminiscent of two famous problems in natural language communication. When very old writings are discovered, in long-dead languages, scholars labor for years over them trying to decipher the original meaning. The receiver, by applying considerable intelligence and ingenuity, can overcome shortcomings in the original message (such as having been written in Etruscan instead of Greek). When a space probe is launched to go beyond the solar system, that space probe includes a document, etched in gold, that is intended to be read and understood by unknown alien civilizations in distant galaxies. The senders of this message spent a considerable amount of time and effort preparing a document whose structure and meaning could be understood by receivers beyond the imagination.

Either the sender or the receiver must take special care, or else the sending and receiving environments must be similar, else the communication of a structured document will fail.

7 Two case studies in successful structured communication

The PostScript graphics language is phenomenally successful at the communication of structured graphics. The Scribe document specification language (and its copies, notably LaTeX[1] is quite successful at the communication of text that has a certain amount of structure. In this section I explain those two systems and offer insights into why they are so successful.

[1] LaTeX is a re-implementation of a subset the Scribe language as TeX macros; for the purposes of this explanation, Scribe and LaTeX can be considered to be equivalent.

7.1 A procedural system: communicating images with PostScript

The PostScript graphics language, implemented by dozens of different laser printers and typesetters from many different manufacturers, was created for the purpose of controlling laser printers [Ado85a]. It is described by its creators as a *page description language.* PostScript is a programming language that is executed by the printer, and can be used to achieve completely device-independent graphics. An image description is prepared as a PostScript program and that program is sent to the printer. The program executes inside the printer, able to interrogate its environment, and as a side effect of that execution, the image is constructed and printed [Rei86b].

A PostScript image representation is able to adapt itself to differing receiving environments because the program that constructs the image is not executed until the details of the receiving environment are known; this permits decisions to be specified by the sender and made properly by the receiver.

For example, a PostScript program can create shades of gray by calling halftone functions inside the printer; these functions take into account the color and density of the ink particles and the behavior of the fuser, and create a certain pattern of dots or lines that will appear to have the correct color of gray. The creator of the PostScript program need not know any of the details of how the abstraction is implemented.

A PostScript program can also make control decisions depending on the execution environment. For example, a PostScript file can be created that will include more text or a more-detailed picture for B4 paper than for A4 paper. A letterhead image can draw the company logo if the printer has high enough resolution, and leave it out for lower-resolution devices, trusting that somebody will put letterhead paper into the printer for the occasion.

As a procedural system, PostScript enables the communication of structured graphical objects by placing the full responsibility on the sender. The sender is given a very complete description of the execution environments that will be available at the receiving end, and must write a program that will anticipate every possible variation in that environment and act appropriately. The receiver will not exercise any intelligence or do any decision-making.

PostScript is so successful because it has done a nearly-perfect job of standardizing the execution environment in which the programs will execute. The language has only existed for 3 years, so it is too soon to tell whether or not the portability of PostScript programs will continue to be as great as it is now.

7.2 A nonprocedural system: communicating documents with Scribe

The Scribe document description language was created for the purpose of making textual documents transferrable between different environments [RW80, Rei80b]. It is a nonprocedural markup language, with associated compiler, by which the abstract form of a structured document can be compiled into an appropriate concrete document. Scribe was designed with the idea that the sender would do as little work as possible, and the receiver would be responsible for making sense out of the document. The source document, created by the sender, is marked up in a very minimal nonprocedural language; the compiler must then expend a large amount of effort performing the binding procedures that are only implied (not explicit) in the structured document source.

The essence of the Scribe document description language is a set of named "environments" that correspond to certain abstract textual effects; for example, the *display* environment is used to put

display text, like this

in the body of a document, and the *enumerate* environment is used to put

1. enumerated lists,
2. like this
3. and this

into the body of a document. There are about 50 standard environments in Scribe, and the bulk of the language design work consisted of the identification and naming of them. the receiver has a wide latitude for interpreting the meaning of the environment names. If the implementation of an *enumerate* environment uses narrow spacing or wide spacing, large letters or small, wide margins or none at all, the meaning is still preserved.

As a nonprocedural system, Scribe enables the communication of structured textual objects by placing most of the responsibility on the receiver. The sender provides a very sparse description of the nature of the document, using terms and tags that are as abstract as possible. The receiver (normally a program) spends a considerable amount of energy making an interpretation of this information that is correct for the receiving environment.

Scribe has existed for 11 years, and imitations have existed for 6 years. Documents prepared for the original 1977 version of Scribe are still fairly usable. It has clearly succeeded at providing a mechanism for communicating documents. The primary reason for its success is also its greatest limitation, namely that there are limits to its extensibility. Whether or not

more extensible systems can be as portable and as durable is an interesting unsolved problem.

8 Kiosk: a system architecture for structured electronic mail

At the DEC Western Research Laboratory in Palo Alto, California, we are working on a new mail system that will enable the reliable and secure exchange of structured documents. This system is named *Kiosk*. We expect to have a prototype version of kiosk working by the end of 1989.

Kiosk will use a network of dedicated message processors, whose only purpose will be the interchange of electronic mail. The message processors themselves are called kiosks. The communication links between kiosks obey an internal protocol that is neither meaningful nor available to the end user.

8.1 Overall structure of Kiosk

A user's computer sends electronic mail by consigning it to the nearest kiosk, just as a person sends postal mail by placing it into the nearest postbox. The kiosks will obey several different protocols for initial receipt of mail: SMTP, UUCP, X.400, and VAXmail. In this regard, the user's computer communicates with the kiosk exactly as if it were the intended recipient. Once the mail has been consigned to the kiosk, it is the property of the kiosk system until it has been delivered to the end user. Usually that will involve moving the mail from one kiosk to another, and then locating the end user at the remote kiosk and exporting the mail to him via whatever export protocol that recipient prefers.

Communication paths between kiosks are encrypted. All traffic in and between kiosks is logged, and no message is deleted from the system until it has been delivered to the end recipient.

Inside the kiosk system, all documents are held in a uniform structured representation called RDIF (Research Document Interchange Format), and kiosk-to-kiosk transfers require the reliable communication of that structure, using the techniques mentioned in Section 5. When mail is received by the initial kiosk, it is translated from the appropriate external form into RDIF. When the mail is delivered from the destination kiosk to the end recipient, it is converted from RDIF into the form desired by the end recipient. Mail programs that have been specifically designed to use RDIF will send and receive mail directly in that format; others will use the import/export protocols.

The heart of the Kiosk system is its internal representation, RDIF, because that representation is what enables it to do automatic conversion and translation of messages. In brief summary, an RDIF document will be a program such that when it is executed by an interpreter inside the receiver's mail reader or document formatter, it will construct the memory data structures necessary to display the desired document.

8.2 Lexical structure of an RDIF document

Syntactically, RDIF uses a very simple, LISP-like linked-list representation. An RDIF document is an S-expression. An S-expression is a list of function calls and their arguments. The semantics of RDIF are based on Scheme, which is to say they are a statically-scoped dialect of LISP. Because the purpose of RDIF is not systems programming but document representation, only those Scheme operators that are vitally important to the ordinary execution of ordinary non-recursive programs are included. An RDIF structured document consists of a header and a body. The header is a single list that contains arbitrarily many sublists and properties; the body is a set of lists, each of which can be arbitrarily complex.

A message that enters the Kiosk system as a text-only message, as for example one that originated in an SMTP mailer, is converted into an RDIF structure in which the body of the message is one list, of which each line is a separate sublist. At the receiving end, upon export, those sublists will be combined into a text file. A message that enters the Kiosk system with a certain amount of structure, e.g. delineated paragraphs, would represent each paragraph as a separate sublist, allowing the export program to handle the paragraph as a unit rather than handling each line separately. A message that enters the Kiosk system with a Scribe or LaTeX structure will be converted into an internal form that is quite similar to the original Scribe document, since most of the Scribe environments have analogues in Kiosk.

The standard import/export form of an RDIF document is virtually identical to the import/export form of a LISP program: a character string using parentheses to denote list structure and quoting conventions to prevent special characters from causing lexical interference. As 25 years of experience with LISP has demonstrated, simple lexical quoting and parenthesis balancing will serve to represent any list structure.

The most important property of the character-form representation in Kiosk is that it is easy to skip over any sublist without examining it, simply by counting parentheses. This means, for example, that if a structured document contains a digitized (facsimile) photograph, and the ultimate de-

livery is to a recipient who has no mechanism for displaying or processing that facsimile image, the software can trivially and reliably skip over that portion of the representation that specifies the image.

8.3 Semantic structure of a Kiosk document

The usual mistake made in interchange standards, such as SGML, is to concentrate on syntax and ignore semantics. This yields a hollow standard, because the semantics of sender and receiver will match up only by accident or by habit, and not by specification or design. The most important part of the RDIF design is the specification of the semantics of the structured message representation. Minimally, RDIF documents have a collection of predefined tag types that correspond to the predefined tag types in systems like Scribe. Additionally, RDIF has an extension mechanism by which new tag definition procedures can be written and incorporated with the body of a document. The procedures are written in the Scheme subset that is the basis of RDIF.

The default, built-in tag type for graphics sub-objects in Kiosk is PostScript. Any sub-object that is marked as being of the appropriate type will be presumed to be a PostScript source program and will be processed as such. If the document is not printed on a PostScript imager, then that sub-object will be ignored or shown as blank.

We expect that RDIF will be used inside DEC for the exchange of mail. Perhaps it will someday be used in other organizations. Where appropriate, the built-in semantics of RDIF can be extended to include "library" definitions that are appropriate for the organization, such as DEC logo and letterhead. When a document using those library definitions is exported in RDIF format, the appropriate definition sublists are added to the exported file. When a document using those library definitions is exported in some other format, the definitions are evaluated in the destination format.

9 Conclusion

Communication of structured objects is fundamentally unsolvable, because structured objects by definition have information missing. That is what makes them structured. If the document is communicated to a recipient who does not have the missing information, he cannot make proper use of it or receive it correctly.

Different schemes for communicating structured objects in spite of these problems place the burden either on the sender or on the receiver. Pro-

cedural systems concentrate the burden on the sender and nonprocedural systems on the receiver.

When a structured document cannot be properly transmitted from one place to another, it is important for the receiver to be able to extract as much usable information from it as possible.

The Kiosk electronic mail system being developed at DEC Western Research is a pilot project to build a mail system based on structured documents and import/export rules. Kiosk uses an internal representation of messages, known as Research Document Interchange Format, that is rich with structure.

Interfaces Between the Designer and the Document

Richard Southall

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
U.S.A.

Introduction

The literature of document production systems is almost entirely concerned with abstract topics: the definition of structure in virtual documents, and the properties of such structures. Questions about the effective realization of these structures in the form of actual objects are seldom considered. This paper attempts to lay some of the groundwork for a discussion of the properties of the actual documents that are the end-product of a document production system's activity.

The paper begins by defining the components of a document's graphic structure, and relates this structure to the visual structure that the reader perceives in the document. In the second section, the relationships between the specifications supplied to the components of the production system at different stages in the production of an actual document are discussed.

The third section starts with a brief description of the characteristics of the logical relationships that exist between the elements of a simple tree-structured document, and the kinds of visual relationships needed to realize them. It goes on to make a comparison between the problems that typographers faced in designing for traditional type-composing techniques and those that a formatting system has to solve when presented with a structured document at its input. The section concludes with a short discussion of the requirements for a meta-design system for typography: a system that could analyze the logical structure of its input, and apply the rules of typographic design in devising an appropriate graphic structure for its realization.

This paper was written while I was working in the Intelligent Systems Laboratory at Xerox Palo Alto Research Center. I am grateful to friends there, and at the Center for the Study of Language and Information at Stanford University, for help in the development of some of the ideas presented in it.

1 Logical, graphic and visual structures in documents

Actual documents are what readers see and use. An actual document is an *assembly of marks on a substrate.* The substrate of an actual document may be *portable* (paper or film, for example); or it may be *fixed*, as when it is part of a hardware display device.

An actual document provides a *visible realization* of a *text.*

Texts are collections of *semantic objects.* The semantic objects in a text have *logical relationships* with one another. The structure of relationships between the semantic objects in a text constitutes the *logical structure* of the text.

1.1 Graphic structures and visual structures

The marks that make up an actual document have *graphic attributes* and *metric relationships.* The graphic attributes of a mark have two components: the *metric properties* of the mark, and its *graphic characteristics.* The metric properties of a mark are such things as *calibre*, *stroke thickness* or *slant.* These are physical properties of the mark itself, and can be measured. The graphic characteristics of a mark are also measureable physical properties of the mark, though they are less easy to measure than its metric properties: they are such things as *edge-sharpness*, *reflection density* and *granularity.*

The *distances between the marks* on the document substrate constitute their metric relationships. Just as the graphic attributes of marks are physical properties of the marks, so the metric relationships between marks are physical properties of the actual document.

In the context of the present discussion, we can make a broad division of the marks in an actual document into two groups: *character images* and *non-verbal marks.* Character images are alphanumeric signs and the symbols used in notational systems. Non-verbal marks are such things as 'rules' and 'bullets'.

The graphic attributes of marks and groups of marks, and the metric relationships between them, constitute the graphic structure of an actual document.

The graphic attributes of the marks in an actual document, interpreted by the human visual system, give the marks their *visual attributes.* The visual attributes of marks are such things as *perceived size*, *weight* (in the typographic sense) or *slope*: in the case of character images, these are the visual attributes that correspond to the metric properties of calibre, stroke thickness and slant that were mentioned above. It is useful to think of

the visual attributes of character images as combining to give the images *typographic attributes* such as *boldness* or *width*.

The metric relationships of the marks in an actual document, also interpreted by the human visual system, give the marks their *visual relationships*. The visual attributes of the marks and groups of marks in an actual document, and the visual relationships between them, constitute the *visual structure* of the document.

Thus the visual structure of an actual document is what the reader perceives when confronted with the document itself. The document has a graphic structure: the reader perceives a visual structure in it.

The reason for drawing out this distinction between the graphic and visual structures of an actual document is that the two are not the same. The visual attributes of marks do not necessarily correspond to their graphic attributes. A simple illustration of this point is the circle-and-square effect familiar to students of lettering. A solid circular shape, placed between two solid square shapes whose sides are equal to the diameter of the circular shape, looks smaller than the squares [AS85]. The three marks have the same calibre, which is a graphic attribute; but the perceived size of the circular shape, which is one of its visual attributes, is different from the perceived size of the square shapes.

Equally, the visual relationships of marks do not necessarily correspond to their metric relationships. There are a number of well-known optical illusions in which parallel straight lines appear to be curved, or lines of the same length appear to be different lengths. An understanding of such optical effects is important in typeface design [LG16, ch. 5].

Notice that visual structure is an essential feature of actual documents. To realize a text, an actual document has to contain marks: by reason of their existence, the marks necessarily have graphic attributes and metric relationships. Even in a document containing only a single mark, the mark has graphic attributes of its own, and metric relationships with the boundaries of the document substrate. In the course of being perceived by a reader, the graphic attributes and metric relationships of the marks are necessarily translated into visual structure.

I take it as an axiom that *the visual structure of an actual document should reflect the logical structure of the text that the document realizes.* This implies that the logical relationships between the semantic objects in the text should be reflected in the visual relationships between the graphic objects in the document.

2 From the text to the actual document

The kind of computer-based document production system that the following analysis describes is made up of a *data capture system*, a *formatting system* and a *marking device*. The data capture system receives input from an author, usually in the from of keystrokes typed at a display terminal. This input is presented to the formatting system, which processes it and passes the resulting instructions to the marking device. The marking device produces an actual document.

Systems like this work in batch mode, running software packages of the kind that Furuta [Fur88] describes as *pure formatters*. To this extent they are beginning to be a little out of date: present-day development work concentrates mostly on the so-called WYSIWYG systems that Furuta calls *integrated editor/formatters*. Pure formatters, though, provide a cleaner environment than do WYSIWYG systems for a study of the ways in which the logical structures of documents are transformed into graphic structures and thence into visual structures.

2.1 From the text to the marking engine

Logical structures for documents, and the formatting systems that handle them, have been extensively discussed in the literature ([BRV83]; [FSS82]; [Jol88]). We can think of the input to a formatting system such as Scribe or LaTeX as constituting a *structured virtual document*, in which the semantic objects in the text are identified and delimited. (This structured virtual document corresponds to Furuta's *document model*.) The formatting system, presented with a structured virtual document, produces from it a *formatted virtual document* (Furuta's *output model*). It is worth while to consider in detail the relationship that the formatted virtual document bears to the actual document that is the end-product of the document production system's activity.

The actual document is produced by the marking device. A marking device contains a *marking engine* and a *device driver*. The engine makes marks on the document substrate according to instructions given to it by the driver. The driver derives the instructions it gives to the marking engine from two sources.

The first of these is the formatted virtual document that the formatting system produces from the structured virtual document. The formatted virtual document is a specification for an idealized version of the actual document. The subject-matter of this specification is the identities, attributes, posi-

tions and metric properties of the *graphic elements* that go to make up the ideal document. These are all expressed in terms that are intended to be more or less independent of the characteristics of any device on which real versions of the ideal document might be produced. (In the TEX world, the `dvi` file is the formatted virtual document. Utting's 1985 paper [Utt85] is an interesting discussion of what 'device independence' really means in TEX.)

The marking device's driver interprets the specification in the formatted virtual document into commands that the marking engine can understand. In particular, the driver converts resolution-independent specifications for the positions of graphic elements in the ideal document into instructions, given in terms of the marking engine's positional resolution, for the positioning of the corresponding *graphic objects* (character images and non-verbal marks) in the actual document.

In the TEX world, at least, the specification in the formatted virtual document contains no explicit information about the shapes of the character images the actual document is intended to contain. The formatted virtual document specifies the position in the ideal document at which it wishes each character image to be placed. The typographic attributes of the image are specified implicitly, by specifying the *font* from which the image is to be produced and its identifying code within the font.

The fonts are the other source from which the driver derives the instructions it gives to the marking engine. Fonts are sets of specifications for the shapes of character images. In present-day technology, fonts can be of two kinds: resolution-specific pixel specifications, or resolution-independent outline specifications. In the first case, the driver can translate the specifications in the font directly into instructions to the marking engine. In the second, the driver has to digitize the shape specified in the font in order to produce the instructions the engine needs.

2.2 From the marking engine to the actual document

Present-day raster-scan marking devices contain actual or virtual pixel arrays, in which each element of the array corresponds to an addressable point on the document substrate. The driver in the marking device changes the states of elements in the pixel array, and the marking engine makes marks on the substrate that correspond as closely as the engine's capabilities permit to the states of the elements in the array.

The state of the pixel array thus constitutes a detailed specification for the positions and shapes of the marks on the document substrate, and hence for the graphic structure of the actual document. The extent to which this

specification is met by the marks that do in fact occur in the document depends on the extent to which the marking engine's performance corresponds to that of the device envisaged by the driver.

The performance of real marking engines departs from ideality in two principal ways. One of these concerns the characteristics of the marking process itself: the other the characteristics of the geometry in terms of which the document substrate is marked. The effect of the first of these can be illustrated by comparing the images produced from identical pixel arrays by electrographic marking engines that 'write black', charging only the marked areas of an initially uncharged surface, and those that 'write white', discharging all but the marked areas of a surface that is initially charged all over. The effect of the second can most easily be seen by looking at the way in which the apparent size, aspect ratio and interlinear spacing of the character images in a display window change when the window is moved around on the screen of a 'high-performance' cathode-ray tube display terminal.

The notion of a marking device's *rendering characteristics* provides a useful way of summarizing the various departures from ideality in the device's performance. Thus one can say, for example, that marking devices containing 'write-black' electrographic marking engines produce better renderings of fine lines than those with 'write-white' engines do; or that devices with high positional resolution are better at rendering small differences in position between graphic objects than low-resolution devices are. Similarly, one can say that non-Euclidean marking geometries imply that the way in which the device renders the metric properties and relationships of graphic objects changes from place to place on the document substrate.

2.3 From the specification to the mark

The passage from the formatted virtual document to the actual document takes place in two stages, at each of which a specification that assumes ideality in some respects is interpreted by a system whose performance in those respects is less than ideal. The instructions to the marking device contained in the formatted virtual document envisage a device with ideal marking geometry and effectively infinite positional resolution. The instructions to the marking engine contained in the pixel array evidently envisage an engine with limited positional resolution, and may also take into account some characteristics of the marking process the engine uses; but they do not normally allow for any departures from ideality in its marking geometry.

The same formatted virtual document, interpreted by marking devices

with different rendering characteristics, may consequently give rise to actual documents with considerable differences in their graphic structure. Because of these differences, readers may perceive visual structures in the different documents that differ enough to give rise to differing interpretations of the logical structure of the text.

3 Designing documents

The classic tree-structured document made up only of chapters, sections, subsections and paragraphs may not occur very often in real life [Nor87], but it provides a useful basis for considering the relationships that occur within and between elements in structures of this kind.

3.1 Logical and visual relationships in actual documents

The logical relationships between the elements of a tree-structured document have the following characteristics:

> *hierarchy*: ranks can be assigned to the elements of the structure.
>
> *containment*: higher-ranking elements contain lower-ranking elements.
>
> *sequence*: elements of equal rank follow one another in the text.

The components of the elements at each level in the hierarchy have logical relationships that are very similar:

> *heading–body relationships*: each element contains a heading and a body. The heading of the element relates more closely to the body of its own element than it does to the bodies of other elements.
>
> *body component–body component relationships*: The body of each element contains lower-ranking elements. These relate more closely to each other than they do to objects outside the element. The body components of an element may themselves have headings and bodies, so that the element has a nested structure: one consequence of this is that there may be a descending sequence of headings at the beginning of an element.

These logical relationships between the elements of the text need to be clearly expressed in the visual structure of the actual document. The visual attributes of the graphic objects that realize the elements of the text should make clear which level in the logical hierarchy each element belongs to. The relationships of containment that exist between logical elements in the text should be realized by the visual relationships between the graphic

objects that realize the elements. The sequence in which graphic objects are perceived in the document should realize the sequence in which the corresponding logical elements occur in the text.

The tools available to the designer for expressing these relationships in the actual document are simple, though their application is not. They are the *space* that separates graphic objects; the *size* of the marks that go to make up the objects; and the *typographic colour* and *style* of the character images that are used. (The typographic colour of character images is a consequence of the typographic attributes of boldness and width that the images possess. Their style is a consequence of visual attributes such as slope, as well as of attributes that have to do with features of their shapes.)

3.2 Document design and traditional typography

Typographic designers' work has traditionally had two parts. One part has been to conceive of visual structures for documents: to devise forms and relationships for the graphic objects that occur in an actual document, so that they solve the problems of expressing the logical structure of a text. The other part of the work, which became more important with the development of mechanical type-composing techniques at the end of the last century, has been to make specifications that would cause a composing system to produce the graphic objects that are required for the first part.

In traditional typography, documents were almost always designed *a posteriori*. Semantic objects that expressed an author's thoughts already existed, embodied in a manuscript (which was an actual document). The boundaries of logical elements in the text were usually marked in the manuscript: if they were not, part of the designer's work was to deduce the logical structure of the text from its semantic structure. The designer's task was to reproduce the logical structure and semantic content of the manuscript in a visually unambiguous and technically practicable graphic form. The nature of traditional marking technologies meant that most of the design work had to be completed before the work of composition could begin.

With present-day document production systems, and particularly with 'generalized markup' systems, the designer has to approach the problem of devising visual structure from a different direction. Authors using such systems are asked simply to identify and delimit elements in the logical structure of the text. The problem of giving appropriate graphic structures to the actual documents that are to be produced is left to the formatting system to resolve.

At the input stage, the analogy with traditional design methods looks fairly close: the marked-up virtual document that results form the author's work is after all not so different from a marked-up manuscript. The difference comes at the next stage: the document production system is expected to reproduce the author's text straight away, as an actual document with a graphic structure that correctly realizes the logical structure of the text.

In principle, there are two ways of resolving the problem that this expectation presents to the document production system. One solution is to build a formatting system that will do typographic design: one that can analyze the logical structure of its input, and devise and specify an appropriate graphic structure for the actual document.

The other solution to the document production system's problem is to supply it with ready-made specifications for graphic objects that correspond to the logical elements the author has marked in the text.

Systems of the first kind would be able to solve the principal problem that generalized markup input systems pose to a formatting system: that authors using such systems have a great deal of freedom in specifying the logical structures of document types. A formatting system that could analyze the logical structure of its input would be able to devise its own repertory of graphic objects to suit the logical elements it found there: it would place no *a priori* limitation on the number of different kinds of logical element an author could invent or use. In systems of the second kind, on the other hand, the repertory of logical elements for which the specifications of corresponding graphic objects exist is necessarily limited, though it may be large. The only way the system can respond to the occurrence in its input of a logical element outside its repertory is by ignoring it: and this response may not satisfy an author whose needs are not exactly met by any of the logical elements that do exist in the repertory.

Another difference between the two kinds of system is in the way they handle the graphic problems presented by the occurrence of differing combinations of logical elements in the input. In order to preserve the visual relationships that correspond to the logical relationships between elements in the text, the graphic form in which a given logical element is realized often needs to differ according to the context in which the element occurs. A simple example is in the heading structures of the present paper: subsection headings that follow section headings have less space above them that subsection headings that occur within a section body.[1] Nested list structures, where some of

[1]This statement was true of the paper in the form in which it was originally realized for the

the items in a list themselves contain lists, present another example: the graphic problems of expressing the logical relationships between the items in the structure can be very difficult to solve.

In order to deal with such problems satisfactorily, the formatting system has to be able to understand the effect of context on the graphic structure that should be produced from a particular sequence of logical elements. This involves altering the specifications for the objects in the graphic structure according to the sequence in which they occur. The graphic problem is not solved by simply concatenating graphic objects whose properties are independent of the properties of their neighbours.

Typographers solve problems of this kind in two stages. They try to discover all the combinations of logical elements that occur in a document, and specify appropriate graphic structures for each combination; and they change their specifications in cases where the visual structure of a particular combination of elements turns out to have been ineffective in expressing their logical relationships. This process of iterative, *post hoc* development of a specification, based on the appearance of the result, is as important in typographic design as it is in type design (see [Sou86]).

In fact the analogy between type design systems and typographic design systems can be followed further. The 'formatting system that will do typographic design', proposed above, would be a meta-design system for typography: one that contained algorithmic expressions of the rules by which typographic design is done, that it could apply to particular problems in the realization of logical structure. This is very much like Knuth's idea of Metafont as a meta-design system for type design, that could capture 'the "intelligence" that lies behind a design' ([Knu79b, p. 17]; [Knu85a, p. 38]).

3.3 A meta-design system for typography: requirements and problems

A meta-design system for typography, supplied with a structured virtual document in which the logical elements of the text were identified and delimited, would generate specifications for graphic objects whose relationships gave rise to a visual structure that realized the logical structure of the text.

In typographic design, the visual relationships between the graphic objects in a document are organized so as to realize the logical relationships between the elements of a text. The logical relationships in the text have semantic significance, and this significance is handed on to the visual relationships

Aussois school. With the present format, the appearance of the graphic structure that results when a subsection heading occurs immediately after a section heading is so unfortunate that the logical structure of the paper has been reorganized to eliminate it.

that realize them. The visual relationships between graphic objects are governed by the size and typographic colour of the marks in the objects, and by the spaces between the objects; but they are also influenced by the visual context within which the objects occur. To take a simple example: the amount of extra space needed after a subsection heading in order to make a clear visual separation between the heading and the paragraph of text that follows it depends partly on the size of the character images in the heading, partly on its extent, partly on the amount of space ahead of it and partly on the amount of space between the rows of characters in the paragraph that follows. All these factors interact in forming the visual structure that the reader perceives in the graphic structure of heading and paragraph.

The visual structure presented to a reader by a single visual unit of a document (which in traditional typography was usually a pair of facing pages, but in present-day documents is more often a single letter-sized page) is a consequence of all the visual interactions between graphic objects that occur in the reader's perception of the unit, and of the reader's understanding of the document-wide visual context in which these interactions occur. The space between rows of character images in a paragraph of text and the spaces around headings are not independent variables in the specification of the graphic structure of the document: the relationship between their values should be determined partly by the way in which they interact in the reader's perception of the structure. An effective meta-design system would know the rules that govern such interactions, and the way they are affected by the values of all the other variables in the design.

Two major types of problem were encountered in an attempt to build an effective meta-design system for typefaces using Metafont, and it seems likely that a meta-design system for typography would encounter problems of the same type. The first was that graphic designers' knowledge of their craft is visual rather than verbal. They know that their work is governed by rules, and they know what the rules are: but they cannot *say* what they are (*cf.* [Sou85, section 4]). The other problem was that, seen from the designer's point of view, the subject-matter of the design system was the wrong one: Metafont allowed the designer to specify and manipulate the *shapes* of characters, when what was needed was a way of controlling the *appearance* of the character images the design system produced [Sou86].

We do not yet know enough about either the mechanisms of reading or the characteristics of the human visual system to be able to make precise numerical statements of the ways in which the visual interactions between

the objects in a graphic structure combine to produce the visual structure that the reader perceives in a document page. What knowledge there is about these interactions has to come from studying the practice of graphic designers. However inarticulate their understanding may be of the rules by which they work, typographers do at least know how to apply them.

It seems to me (speaking here as a typographer, and perhaps overgeneralizing from my own experience) that there are three things to be concluded from such a study. Firstly, the primary tool with which typographers work is the *space* around graphic objects. Secondly, they use space in very small increments: differences in spacing of a few tenths of millimetres vertically, or a few hundredths horizontally, make substantial differences to the visual relationships between objects. Thirdly, and most importantly for our purposes, typographers' decisions about space are made on the basis of *the appearance of the actual document* they are dealing with, rather than in terms of the abstract properties of a metrical structure.

The typographer and the type designer, trying to specify rules for their respective meta-design systems, are thus faced with exactly the same problem. They are asked to make precise *a priori* statements of general rules that they can only discover by *a posteriori* observation of particular instances of their application. It is easy to say 'There should be more space after a subsection heading than between the rows of characters in the paragraph that follows it', just as it is easy to say 'The two parts of the vertical stroke of small *h* should be slightly different in weight'. The questions 'How *much* more?' 'How *much* different?' cannot be answered correctly, in the present state of our knowledge of the human perceptual system, except by reference to the actual objects that the design system produces.

4 Conclusions

An understanding of the logical structure of a text is essential to a reader's comprehension of the text. This understanding comes to the reader from the visual structure of an actual document, which is a consequence of the actual graphic structure that the document presents to the reader.

The actual graphic structure of documents is not fully specified by the formatted virtual document that a formatting system produces from a structured virtual document: it is also a consequence of the rendering characteristics of the marking device by which the document is produced. Thus different marking devices produce differing actual graphic structures from the same formatted virtual document.

Generalized markup systems that allow authors to devise their own repertories of logical elements for structuring a document need to be supported by meta-design systems that specify graphic structures for a document as a consequence of analyzing the graphic structure of its text. Systems that work by one-to-one allocation of graphic specifications to logical elements cannot be relied on to handle correctly either arbitrary sequences of logical elements or the input from generalized markup systems.

In the present state of our knowledge, we cannot make explicit statements of the rules by which designers assign graphic structure to the logical structure of a text. This fact, and its consequences for the development of document production systems, seem to suggest a useful direction for future collaborative work between semanticists, psychologists, graphic designers and computer scientists.

Text Structure Recognition in Optical Reading

Rolf Ingold

Ecole Polytechnique Fédérale de Lausanne
(Swiss Federal Institute of Technology at Lausanne)
CH-1015 Lausanne (Switzerland)

ABSTRACT
This article treats the process of reading a printed text as the inverse function of formatting. Such an approach requires precise definitions of various structures. We start by considering the problem of the recognition of typographic structure. We then show that a logical structure can be deduced from it based on a formal description of the document's style.

1 Introduction

The aim of optical reading is the automatic input of texts from their printed form on paper. A camera or scanner reads a page of text as an image, usually in two levels (white, black). This grid of pixels is then processed and analysed by recognition software.

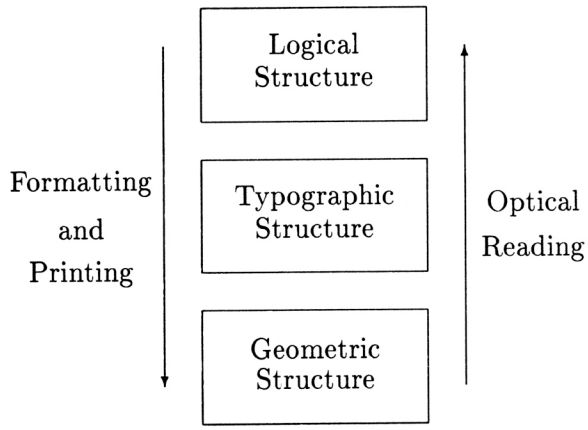
Optical reading has many aspects and the means used are very varied [GB79, Man86]. The areas in which most work has been done are undoubtedly reading of printed or typewritten texts [FF62, Hol76, LB85]. But more recently work has been carried out on the recognition of Arab and Chinese alphabets, or on manuscripts [SBM80].

However, one field has been very little studied so far: the recognition of structures within a text. Methods have been developed [RS86] to detect graphic figures and pictures, allowing text portions to be distinguished. Some rare studies [VV73, NSS85] propose a model of structures based on the location of characters. However, the information that can be deduced from the choice of a particular font is generally ignored. Current systems are exclusively concerned with the recognition of characters, with no account taken of their physical representation (font and size).

Structure recognition does, however, have at least two advantages:

1. automatic input and recording of databases from printed archive material;
2. incorporation of optical character recognition within a complete document production and handling system. From this point of view,

Figure 1: The various structures of a text for optical reading



optical reading has to be regarded as the inverse transformation of formatting and printing.

This article provides a preliminary view of the issues involved, based on a study carried out at the *Ecole Polytechnique Fédérale* at Lausanne, on the optical reading of legal texts. It will only consider structural recognition for phototypeset documents, i.e. documents of good typographic quality.

Section 2 highlights the various structures within a text. Section 3 presents methods for recognizing and interpreting these structures. Finally, section 4 gives an abstract model for the description of structures.

2 Various structures of a text

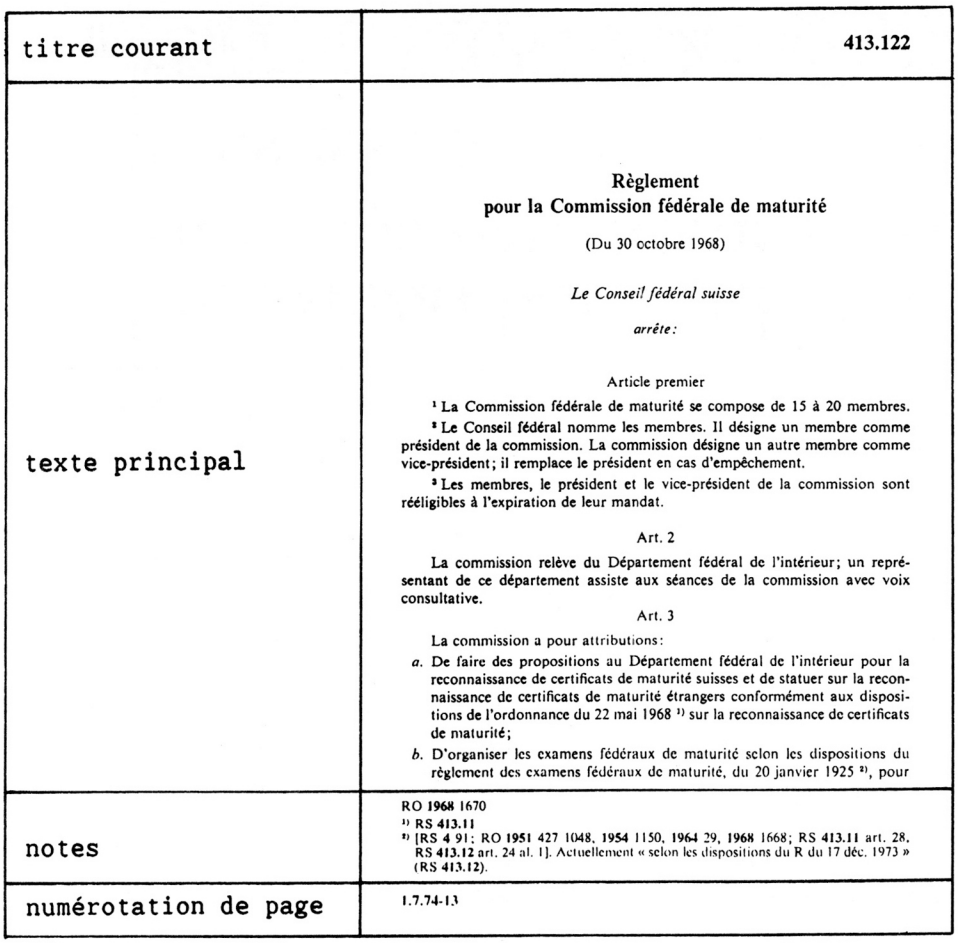
The structure of a text is a relative concept. We describe the main structures involved in optical reading of phototypeset texts in this section (figure 1).

2.1 Logical structure

The logical structure of a text is, in principle, determined by the author. It breaks the text down, for example, into chapters, sections and paragraphs, it defines headings and it determines links and references between various elements. The structure in question is wholly abstract and totally independent of the way the text is presented. It establishes links between the various parts of the text.

The logical structure of a text is often represented by a tree [CIV86, Fur86a, QV86, QVB86a, QVB87]. This model is particularly useful since it allows both sequencing and hierarchical nesting of elements to be expressed.

Figure 2: Breakdown of a page into regions

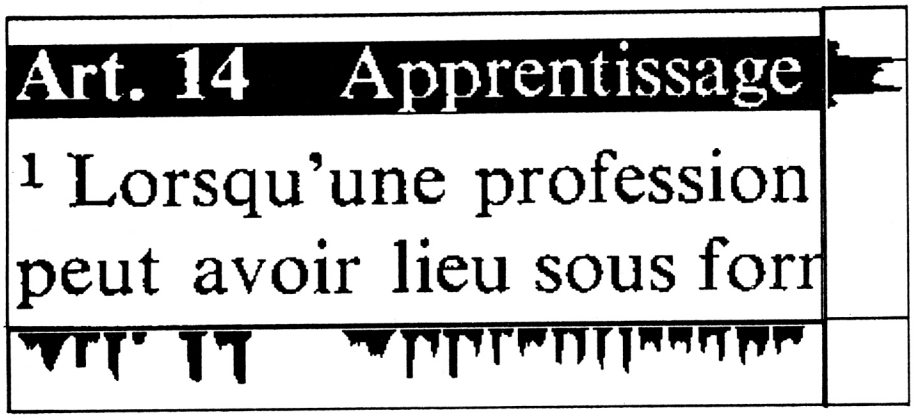


2.2 Typographic structure

This structure characterizes the presentation of the various text elements in terms of blocks, independently of their page layout. It ignores the concept of lines, since these are a result of the arbitrary decomposition operation carried out during formatting. On the other hand, it does describe the alignment option selected (ranged left, ranged right, centred, justified) for text elements.

Typographic structure also describes the fonts used for the various text elements. Several models can be applied: the simplest is to specify one font

Figure 3: Projection of a line of text



from a list of possible fonts. It is also possible to consider a triplet (style, size, weight) as the determining characteristic of a font.

2.3 Geometric structure

This structure expresses the way in which characters are actually laid out on the printed page. One model would be a breakdown of the page surface into regions, themselves made up of several lines, according to the following criteria:

A region is made up of a set of lines with an unambiguous sequence. This allows a distinction to be made, for example, between the main area of a page, the marginal notes area, a possible footnote area and the running head position.

A line is considered as a sequence of characters placed on a base line, and is characterized by the upper and lower limits of the line together with its left and right margins.

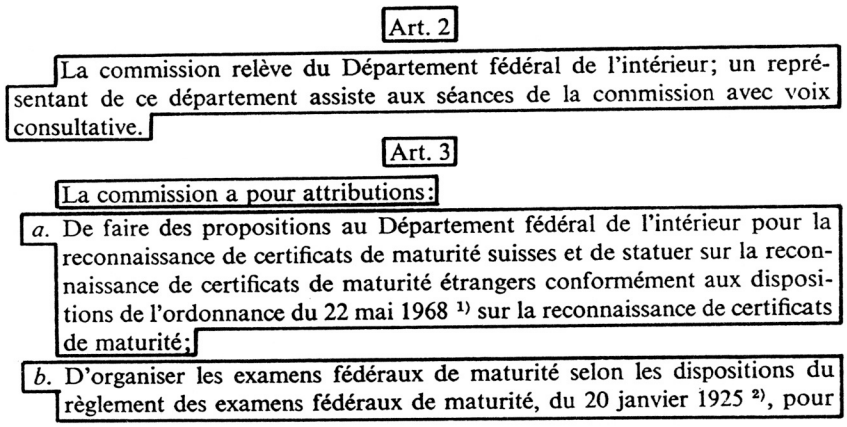
3 Structure recognition

This section is concerned with a study of the links between structures defined in section 2 and the ways in which each in turn (geometric structure, typographic structure and logical structure) can be recognized, within the context of optical reading.

3.1 Recognition of geometric structure

The aim at this stage is to break the page down into regions and lines.

Figure 4: Typographic structure obtained by grouping of lines



3.1.1 Breakdown of the page into regions

Delimiting the regions on a page is a delicate problem as the criteria are by no means absolute. In certain cases, it is possible to apply rules based on the position of the text, the fonts used or the contents. In general, manual intervention by an operator is necessary to define regions (figure 2).

3.1.2 Breakdown of regions into lines

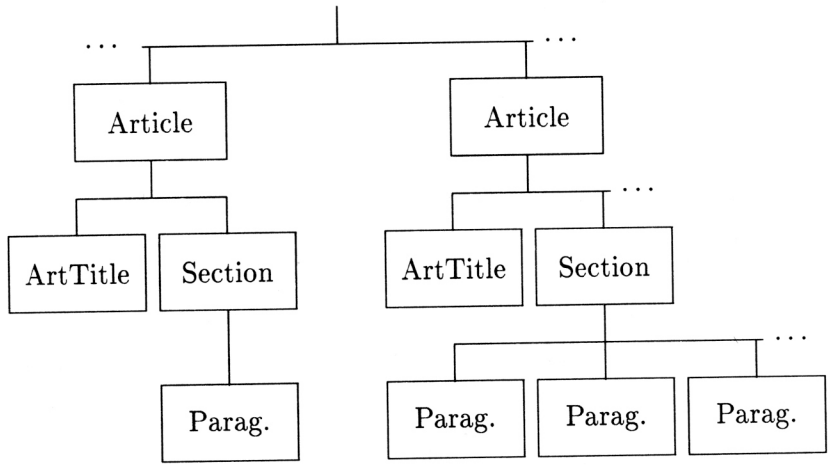
Once regions have been determined, we have in turn to break them down into lines. Known methods for image segmentation can be applied here. These methods are based in particular on projections of a block of the image on to horizontal and vertical axis. The main problems met are those of bad alignment of the image; this is not, however, a subject that falls within the scope of this article.

3.2 Identification of characters

The current trend in character identification is to use typological or syntactic character criteria [KPB87]. These methods have the advantage that they are not sensitive to deformations, and apply successfully to all typographic fonts with the possible exception of certain stylized characters.

The recognition of a typographic structure within a text, however, has to use an identification algorithm which is sensitive to font. Global comparison methods, point by point, seem the most appropriate for that problem.

Figure 5: Logical structure induced by the typographic structure



3.3 Font identification

This stage is necessary in order to recognize the typographic structure of the text; two extreme solutions can be examined:

1. a single operation to identify the character and font used;
2. *a priori* identification of the fonts over whole regions, lines or words.

The first method seems the most robust; it nonetheless has the disadvantage of requiring a vast number of comparisons, since the number of characters to distinguish is multiplied by the number of fonts to consider. Moreover, since changes in font are relatively infrequent, the second method seems more promising. This approach is corroborated by the fact that the human eye seems to proceed in the same way: it distinguishes fonts (italics or bold) without deciphering characters.

The processes that are currently being studied to carry out font identification in a homogeneous region use the following criteria:

- the height of a line;
- the density of dots;
- the distribution of black areas;
- the spectrum of the projection of the text onto a horizontal line (figure 3) and its derivative;
- the correlation between various lines of points;
- etc.

These criteria alone are not sufficient to identify the font unambiguously. But the identification of a character by global masking in principle allows this ambiguity to be overcome.

3.4 Recognition of typographic structure

The issue here is to delimit lines and sets of lines forming uniform entities, and determine the alignment or justification mode used for those entities.

3.4.1 Line grouping

Grouping lines together (figure 4) can be based on homogeneity in fonts and the line spacing regarded as standard for the main font. The position of margins can, in certain cases, be used to confirm or deny a hypothesis.

In practice, a difficulty occurs in the relatively exceptional case in which changes of fonts within a continuous text coincide with line breaks.

3.4.2 Determining the mode of alignment

The mode of alignment is deduced, in principle, from the positioning of the margins of each line. In the case of grouped lines, determining the mode of alignment is reasonably easy.

Individual lines are more difficult, since there is no context with which to compare their position. For example, it is a priori impossible to decide whether a line is centred within a page or aligned on a locally indented margin.

3.5 Recognition of logical structure

This can be regarded as the final aim and the last stage of text structure recognition. It is based on an interpretation of the typographic structure based on precise rules. Figure 5 presents the logical structure that can be deduced from figure 4.

These rules vary as a function of the documents analysed and have to be adapted to suit them. They therefore need to be expressed in an appropriate form for a particular class of document.

4 Structure description languages

A formal description is necessary in order to parameterize recognition algorithms and, in particular, to interpret the typographical structure in order to build up the logical structure. This section describes the main characteristics required for this formalism.

Figure 6: Description of a logical structure

```
<Proceedings>= MainTitle [SubTitle] [Date] [Authority]
               {Basis} [Endofpreamble]
               {[PartTitle] {<Article>}}.

<Article> = ArtTitle { <Section>}.

<Section> = {Paragraph}.
```

To meet its aim of easing structure specification, the formalism proposed is descriptive and has two distinct parts.

4.1 Description of logical structure

A formalism based on formal grammars seems appropriate to describe a generic logical structure. The non-terminals of this grammar represent the various hierarchical objects in the logical structure, while terminals correspond to blocks of the typographic structure.

Rewrite or construction rules (depending on the point of view adopted) allow optional objects to be generated, as well as sequences, alternatives and iterations (figure 6).

4.2 Description of typographic form

The issue here is to find a means of associating a text element with a terminal concept in the logical structure. A table of characteristics can be used (figure 7). For each terminal concept in the grammar, it contains a set of conditions which all have to be met for the association to be established.

Conditions are expressed as a function of the following information associated with text elements:

- contents, i.e. a character string;
- the set of fonts used
- the number of lines, their margins and their assumed mode of alignment.

Figure 7: Description of a typographic form

```
Main Title:  Alignment = centred
             Font = (Times, Bold, >12);

Date:        Alignment = centred
             Contents = "(" + {?} + ")",
             Font = (Times, =, 10),
             lines = 1;
...

Section =  Content = figure {figure} + {?}
...
```

5 Conclusion

Research in the field of optical reading of texts has been going on for some 20 years. On the other hand, recognition of text structures has only begun to be examined very recently [SZ86]. The advantages of this approach are, however, obvious: automatic input of texts for databases, inclusion of optical reading in a structured document handling system, etc.

This article describes one approach to the problem for typeset documents. It proposes a decomposition of the problem into several stages based on the geometric structure, the typographic structure and finally the logical structure of a text.

The variety of document styles to recognize requires a formalism to express document presentation rules. The characteristics of such a language have been briefly sketched here.

Despite reasonably precise formalization of the problem of recognizing text structures, certain difficulties cannot be overcome. This is a consequence, most notably, of the non-algorithmic nature of text formatting and printing. Methods can, of course, be improved, but we cannot rule out the need for intervention by an expert system, or even by an expert human, in order to solve the more difficult problems.

Semantic Structures of Text

Pat Norrish

Agricultural Extension and Rural Development Centre
University of Reading
London Road
Reading RG1 5AQ
U.K.

1 Introduction

"Text . . . has a semantic base (meaning) and is organised into units (e.g. words, sentences, lists, headings, equations and formulae). All texts are composed of one or more such units. The number and kind of units and the ways in which they are related to each other constitute the structure of a specific text" [Nor87]. As the proceedings reported in this volume show, such structures are of vital interest to the computing community now that documents can be produced on computer-based systems necessitating what Southall calls the 'a priori' approach to document design [Sou84]

Traditionally the design of documents was carried out after the document was written, by people highly trained and skilled in editing, designing and preparing manuscripts for production. Computer-based systems however give the originator the opportunity to take the document through all the stages needed for production. This causes problems, as originators are usually not trained in editing and design skills and have rarely had to produce a great variety of realizations of text structures, although they are able to use and understand them. The models of text which are offered to those originating and producing documents on computer-based systems are criticized by typographers for being over-simplistic and appearing to pay scant attention to the appearance of real documents and the needs of the user. The typographer's worry is that the clarity, consistency and diversity with which structures can be realized will be lost and the reader will suffer in consequence.

Readers have seemingly conflicting needs for both diversity and consistency in the structure and realization of documents. Diversity to aid identification, location and recognition of a specific document and of the different kinds of information it contains, consistency to help in recognition of, for example, classes of documents and information. While in theory each docu-

ment (and each page within a document) could be unique in structure and realization, if it were so in practice the cognitive load for the reader would be overwhelming. Luckily conventions have developed over the centuries for different kinds of documents, readers, and circumstances of use, and for production on a variety of systems. These conventions are important for the user because of the role that prior knowledge plays in the understanding of written information [Kel86]. This role is demonstrated by Twyman in lectures in which he shows a set of slides of texts presented in highly conventionalized ways. They are shown in rapid succession leaving no time for the audience to read for content; nevertheless, the categories of text shown are always recognised. This demonstration can be further backed up by the results of a small survey carried out by Norrish [Nor87] in which people were asked to list the categories of graphic material, printed or otherwise, which they recognised. The categories listed were very similar from person to person across a range of occupations. Earlier work on the reading of a journal index by Wright and Threfell has shown that where readers are very familiar with a convention a change may cause many errors [WT80].

The conventional use of structures and their realizations is, then, important for the reader and thus for the originator who has to produce them. Gaining a thorough knowledge of what these conventions are, of how and when to use them, and the kinds of decisions involved in their use, is a problem for those who have not been trained in the craft of the traditional printing and publishing industry. There appear to be two ways of tackling this problem. One would be to ensure that all those using computer-based systems are well versed in, at least, the conventions of the material they have to produce. The other would be to provide systems designers and software writers with the kind of knowledge employed within the conventional publishing industry. In either case, analytical and descriptive work would have to be carried out in order to determine what the dimensions of conventional use are.

Some analytical and descriptive work of the kind suggested above has already been carried out as part of a British Library funded project called "The Graphic Translatability of Text". The overall aim of the project (which ran from January 1982 to February 1985) was to look at some of the changes in information skills and technology, with a view to providing a better understanding of the organisation and presentation of texts. A threefold approach was taken to the project: a typographic language survey, a series of case studies, and the testing of a facility (colour, which comes free with some new technologies). What follows here is a brief overview, with some discus-

sion, of the way in which the survey of typographic language was carried out. It is largely taken from the report of the project [Nor87].

2 Approaches to description and analysis

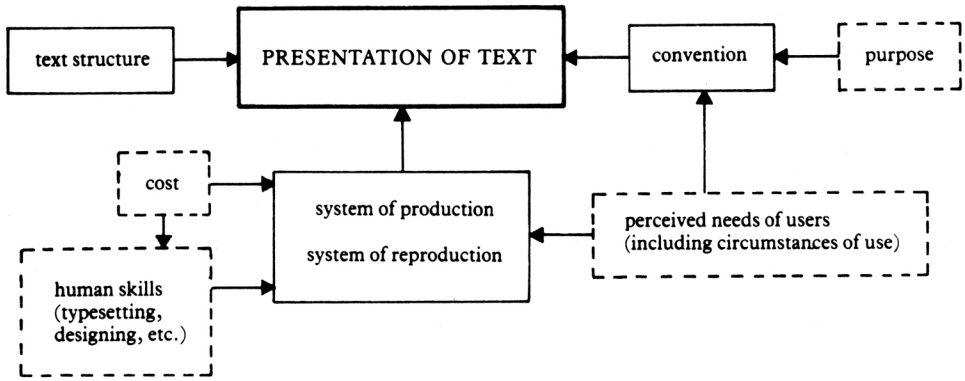
The description and analysis of texts is not new. Bibliographers, for example, carry it out in highly specific detail in order that particular works or editions can be identified. Within the traditional printing and publishing industry, designers and editors carry it out when preparing a manuscript for production.

The latter example appears to be a fruitful area for the study of convention. The decisions of designers and editors are embodied in a set of instructions (specifications) for those who are to compose, make-up, and print the text. Specifications usually consist of the following:

- a set of general written instructions covering such things as the kind, weight, colour and size of paper, the size weights and styles of typefaces, details of the colours of ink to be used etc.
- marked-up copy giving instructions for the realization of all text structures (marking-up is done in coded form using special signs and symbols)
- layouts which back up the marked-up copy and show what particular parts of the text should look like. These may cover general structures like chapter openings and the treatment of footnotes, or unique or particularly complex structures.

Ideally specifications relate to the system on which the text is to be composed and in theory they are complete, rigorous and precise in terms of, for example, measurement. They appear to be a good starting point for the understanding of structures and realizations. However, for various reasons this is not the case. In practice they are often not complete, instructions may not always be written, they may be passed on verbally and changes may be made which are not always transferred to the original specification. In addition the specifications usually disappear once a job is finished, so that getting hold of a wide range of them, and interpreting the special signs and symbols used in mark-up, would not be an easy task. Finally they give very little idea (except in layouts) of what the reader will see and use. To determine the full dimensions of convention (regularity, diversity, exceptional use etc.) normally seen by the reader it is necessary to analyze and describe real documents designed for use in particular circumstances.

Figure 1: Factors that affect the presentation of text



3 Selecting documents

In order to document the conventions of structure and realizations it is first necessary to establish a framework for the selection of documents. This needs to be based on the circumstances under which it might be expected that conventions would vary. This was done for the typographic language survey by taking into account: the intentions and knowledge of typographers about such circumstances; the factors which affect the presentation of text (Figure 1) and the categories revealed by the survey of non-typographers. An overview of the classification scheme which resulted from these considerations is shown in Figure 2. The material chosen for analysis was taken from the information category in two widely differing contexts:

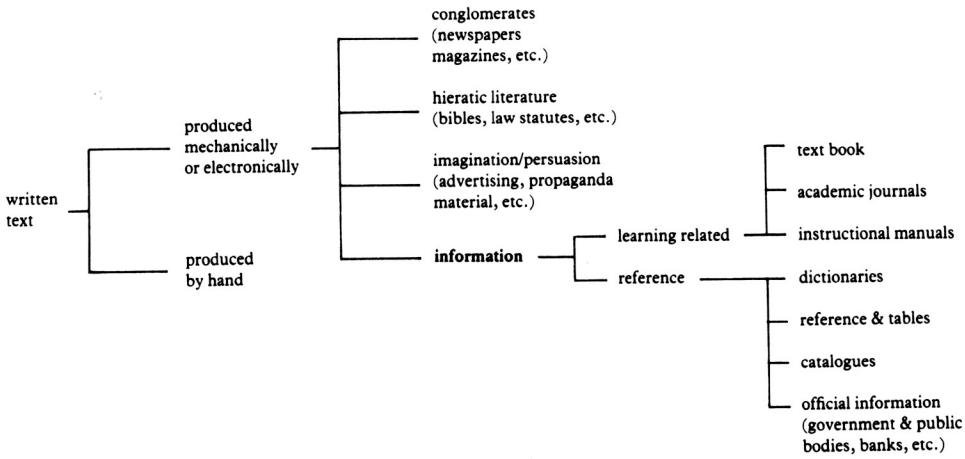
- information directed at, or provided for, the general public by an official or government body (leaflets and booklets)
- information intended for specialist users, written and produced by specialists (academic journals).

4 Analysis and description

The approach used towards analysis and description owes a great deal to work in linguistics where analytical and descriptive work on grammar (structure) and realizations is well established at the level of the sentence and where techniques for examining larger chunks of text are developing rapidly [dBD81, CD69, Hal85].

Text is usually assumed to have a hierarchical structure. For instance a book can be seen as consisting of chapters; chapters of sections; sections of

Figure 2: An overview of the framework for the selection of material



paragraphs; paragraphs of sentences; sentences of clauses; clauses of words; words of characters. As you move up through the levels of the hierarchy the amount of information carried is greater and the relationship of a higher level to a lower one is that of inclusion. However, there are text structures which cannot be accounted for in this way [GP82]. Running heads are one example. When present they appear in the same position on every page of a document, but change their semantic content depending on the subject matter of chapter or section. Running heads are part of the access system for the reader, but unlike headings are not located within the main body of the text. Tables are another, more complex example. Because of their two-dimensional nature it is impossible to describe them in a hierarchical manner. Each item of data is inextricably linked to two different headings (row and column) and to other data items in the same row or column. So although a hierarchical tree structure diagram would enable a description to be made of some texts, for others it would not work. The solution within the project was to make modified tree diagrams with notes for unusual or complex situations. Work was not carried out at the character, word, or sentence level on which some work has already been done [Wal80, Wal83].

Structures were recognised by looking at the document for visual signals and cues which alert the reader to something different happening in the text. As a first step the document as artifact was described using the parameters shown in Figure 3. Once that had been done, a modified tree diagram of the

kind shown in Figure 4 was made. Such diagrams contained the following information:

- the overall structure of the text and the relationship of the component parts to each other and to the whole
- the access system (headings, cues to lists, paragraphs etc.)
- the location of different parts of the structure
- the extent of the text
- the orientation of parts of the text which were different from that of the whole document.

It was originally intended to build descriptions of realizations into the modified tree diagrams. This was to be done by describing each realization of a structure separately and denoting it by a small letter of the alphabet. However, while this worked well for paragraphs it became unworkable as soon as lists were looked at, because within a short space of time forty-seven different list realizations had been described. Descriptions of all except paragraphs were thus kept separate from the tree diagrams.

Because we were trying to document what the reader sees, no precise details of such things as text measure, type style, size or variant, or exact line spacing were recorded. However, changes which signalled that something different was happening in the text were recorded in relative terms. A heading might be larger than text type for example, or it might be in a display size, variants of type might be recorded as heavier than text type, etc.

Descriptions were built up by asking a series of questions about each structure designed to show what signals and cues were being given to the user. The questions shown below are representative of the kinds of questions asked but relate only to paragraphs and tables.

5 Paragraphs

Paragraphs are not usually thought of as graphically complex structures, but as the simple building blocks of text, and initially we asked only the following three questions about them:

1. What is the alignment of the first line in relation to successive lines?
2. If there is more than one paragraph, how are they separated from each other?

Figure 3: Parameters for the description of different forms of documents

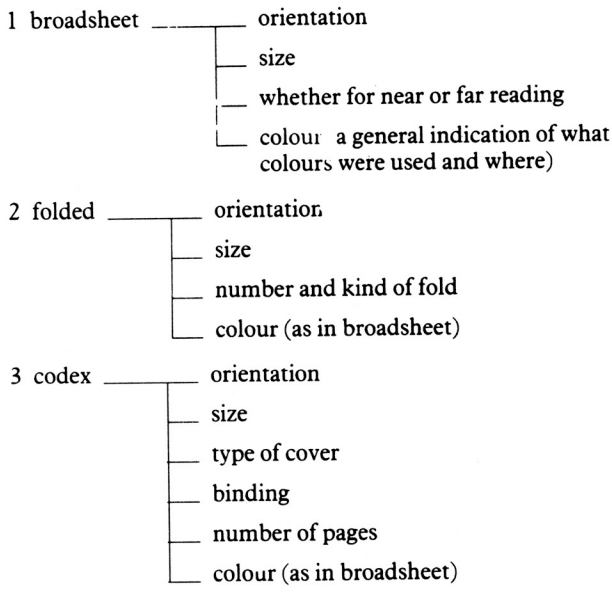


Figure 4: Modified tree diagram

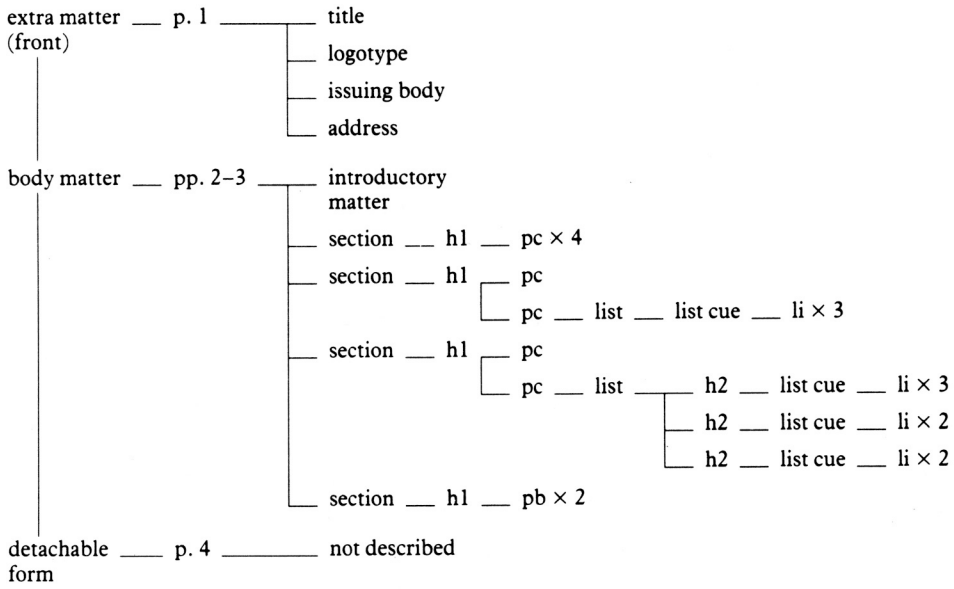
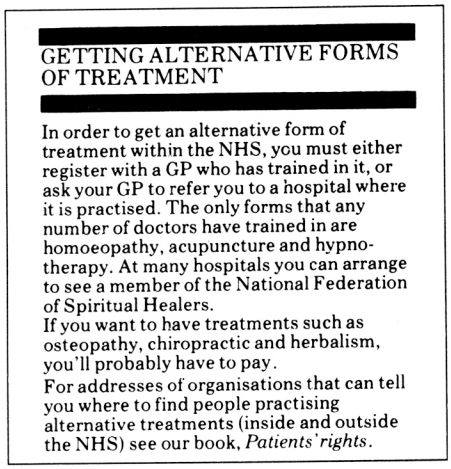


Figure 5: An example of ambiguity in paragraph separation (reproduced by permission of the Consumers Association)



GETTING ALTERNATIVE FORMS OF TREATMENT

In order to get an alternative form of treatment within the NHS, you must either register with a GP who has trained in it, or ask your GP to refer you to a hospital where it is practised. The only forms that any number of doctors have trained in are homoeopathy, acupuncture and hypnotherapy. At many hospitals you can arrange to see a member of the National Federation of Spiritual Healers.

If you want to have treatments such as osteopathy, chiropractic and herbalism, you'll probably have to pay.

For addresses of organisations that can tell you where to find people practising alternative treatments (inside and outside the NHS) see our book, *Patients' rights*.

3. Is the normal convention for marking the start and end of a paragraph adhered to? (Start and end should be coterminous with the start and end of the first and last sentences respectively).

Such questions covered the great majority of paragraphs which tended to be signalled and separated from each other by the use of space in one of three ways:

1. the first line of each paragraph was indented by two or three characters (maybe more)
2. extra space was included between paragraphs, but there was no indentation of the first line
3. the first line of each paragraph was indented and extra line space was used between them.

In one instance none of these methods was used, the space left at the end of the last line of each paragraph obviously being assumed to be enough indication. The resulting ambiguity for the user can be seen in Figure 5.

However, paragraphs are more complex than at first appears. While it is true that many fulfil the notion of visually cohesive, simple structures, there is a second group which interacts with other structural elements in ways which make them visually complex. Paragraphs in leaflets, for example, often contain lists and addresses, and in journals they may contain lists, formulae, equations or quoted matter. For convenience these two major divisions were called solid and interrupted. Maintaining the visual cohesion of

an interrupted paragraph in relation to solid paragraphs while differentiating the interrupting structure is one of the important typographic decisions which have to be taken in relation to all but the simplest of documents.

Paragraphs, whether solid or interrupted, were found to be graphically different from each other in relation to their position in a document, and in relation to different kinds of documents intended for different kinds of readers. For instance initial paragraphs which came immediately after a title or heading were usually different from subsequent paragraphs. In the case of the journals studies this difference was a small one, for example the first line of the initial paragraph would not be indented whereas subsequent first lines would be. In leaflets the differences were much greater. An initial paragraph might, for example, be in a different, larger type size; it might be in colour; if the body matter was in two columns, the paragraph might run across both, and so on (see examples, Figures 6–8). A different kind of complexity arose in some journals in which a series of paragraphs were alphabetically or numerically cued, giving them a list-like appearance. In short, paragraphs were found to be much more complex than they at first appeared.

6 Questions to ask about tables

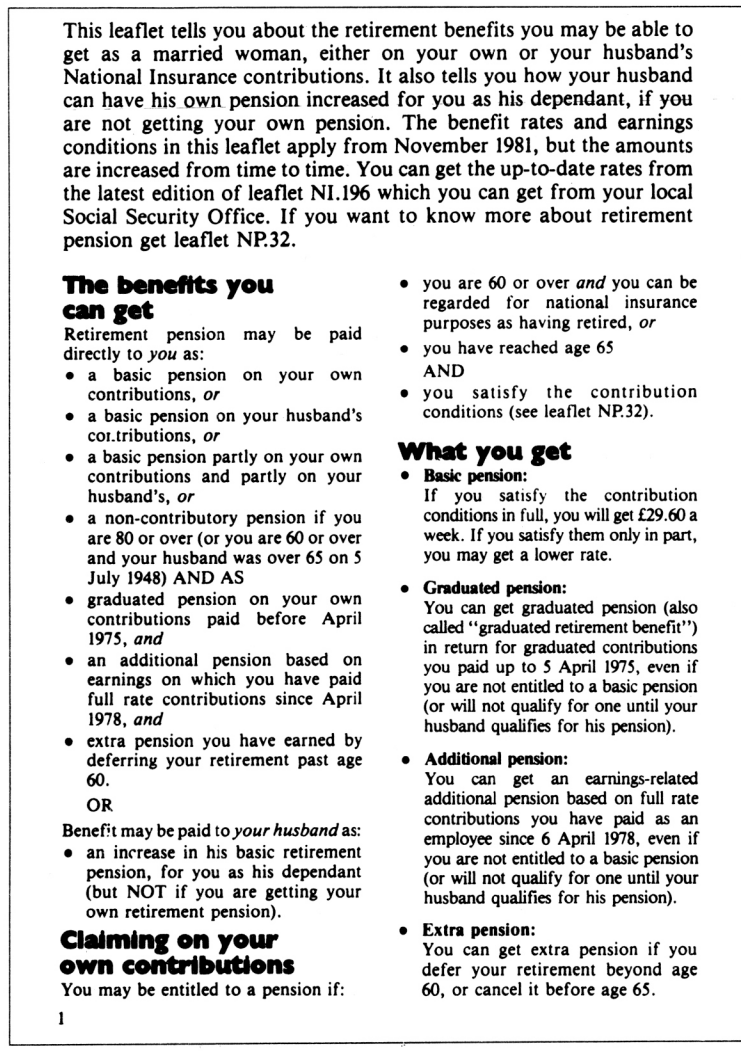
Although most of us think about paragraphs as simple structures, the same cannot be said for tables which are usually perceived as graphically complex and difficult to use. As I have pointed out elsewhere [Nor84] tables are made up of several different kinds of information, each of which has a specific location and relationship to the data contained in the table. These include:

- general text matter about the table
- row and column headings which lead the reader to one specific piece of data in one field of the table
- hierarchical or superordinate headings, which relate to more than one row or column
- data in the field of the table

Tables may also have keys to explain information given in coded form, which may relate to any of the other kinds of information listed above. Finally tables are usually referenced to the text by a table or figure number and a heading.

From the point of view of the user this information can be grouped into two main parts. One is the table body, which contains all levels of row and

Figure 6: An introductory paragraph spanning two columns which is in a different type size from the main text



This leaflet tells you about the retirement benefits you may be able to get as a married woman, either on your own or your husband's National Insurance contributions. It also tells you how your husband can have his own pension increased for you as his dependant, if you are not getting your own pension. The benefit rates and earnings conditions in this leaflet apply from November 1981, but the amounts are increased from time to time. You can get the up-to-date rates from the latest edition of leaflet NI.196 which you can get from your local Social Security Office. If you want to know more about retirement pension get leaflet NP.32.

The benefits you can get

Retirement pension may be paid directly to *you* as:

- a basic pension on your own contributions, *or*
- a basic pension on your husband's contributions, *or*
- a basic pension partly on your own contributions and partly on your husband's, *or*
- a non-contributory pension if you are 80 or over (or you are 60 or over and your husband was over 65 on 5 July 1948) AND AS
- graduated pension on your own contributions paid before April 1975, *and*
- an additional pension based on earnings on which you have paid full rate contributions since April 1978, *and*
- extra pension you have earned by deferring your retirement past age 60.

OR

Benefit may be paid to *your husband* as:

- an increase in his basic retirement pension, for you as his dependant (but NOT if you are getting your own retirement pension).

Claiming on your own contributions

You may be entitled to a pension if:

- you are 60 or over *and* you can be regarded for national insurance purposes as having retired, *or*
- you have reached age 65

AND

- you satisfy the contribution conditions (see leaflet NP.32).

What you get

- **Basic pension:**
  If you satisfy the contribution conditions in full, you will get £29.60 a week. If you satisfy them only in part, you may get a lower rate.
- **Graduated pension:**
  You can get graduated pension (also called "graduated retirement benefit") in return for graduated contributions you paid up to 5 April 1975, even if you are not entitled to a basic pension (or will not qualify for one until your husband qualifies for his pension).
- **Additional pension:**
  You can get an earnings-related additional pension based on full rate contributions you have paid as an employee since 6 April 1978, even if you are not entitled to a basic pension (or will not qualify for one until your husband qualifies for his pension).
- **Extra pension:**
  You can get extra pension if you defer your retirement beyond age 60, or cancel it before age 65.

1

Figure 7: An initial paragraph in an introduction (left-hand column) which is in a different type style and variant and a much larger size than the other paragraphs in the introduction (Crown Copyright; reproduced by permission of the Controller of HMSO)

This is a brief guide to keeping VAT records and completing the VAT return. It is aimed at helping you whether you are new to VAT or have been registered for some time.

However, it cannot tell you all you need to know. There are Notes on the back of the return form, and there are a number of Customs and Excise notices and leaflets which go into more detail. For some of the titles and details of where to find a complete list, see 'Further Information'.

Remember, your local VAT office is always ready to help you.

If you have a problem, don't guess at the answer
ASK FOR ADVICE

Keeping your VAT records

What we require is that your books are complete and up to date and that the figures used to fill in your VAT return are easy to find.

There are many forms in which you may keep your business records. The choice is up to you. This guide gives only a few basic guidelines. For full details see the **VAT General guide** (Notice No.700), Section IV (Tax Invoices) and Section VI (Records and Accounts) and **Special schemes for retailers** (Notice No.727 and Supplements).

Your sales records

How you keep your sales records depends on whether you make out an invoice for every sale.

If you sell to customers who, like yourself, are registered for VAT you must give them tax invoices so that they can reclaim VAT due to them. Keep a copy of each full invoice you issue. If you are a retailer, you need not keep copies of less detailed tax invoices which you may issue for sales up to £50 (including VAT).

When you list your invoices keep a separate column for VAT. This VAT is your 'output' tax.

If you use a **retail scheme** you can work out your output tax without recording each sale but you will need to keep a record of your total takings each day. When you count the money in your till, remember to include sales against credit cards etc. and to add any amount taken out during the day.

Whether or not you use a retail scheme, don't forget that VAT is also due on:

- sales to your staff
- gifts of goods costing over £10
- goods taken for private use
- the full value of goods sold where something has been taken in part exchange
- sales of assets such as office equipment
- any other taxable business income, even from a sideline.

Figure 8: An example of an introductory paragraph in a different type style, size and variant from the text type (Reproduced by permission of the Department for National Savings)

Twenty-Fifth Issue National Savings Certificates are for longer term savings. If you want the benefits of a guaranteed return – entirely tax-free – these Certificates could be just what you are looking for.

What is the interest rate?

The return is equivalent to a compound annual interest rate of 7.51% over the full five years. As an example, a £100 purchase would, after five years, increase to £143.60.

Are they tax-free?

A big attraction for taxpayers is that this yield is tax-free. Repayments of these Certificates are free of all UK income tax (including investment income surcharge) and capital gains tax. They are also free of income tax in the Channel Islands and the Isle of Man.

How much do you get back?

Each unit costs £25 and becomes £35.90 in five years. Here are some examples:

£100 (four units) becomes £143.60 in five years.
£500 (20 units) becomes £718.00 in five years.
£1000 (40 units) becomes £1436 in five years.
£2500 (100 units) becomes £3590 in five years.
£5000 (200 units) becomes £7180 in five years.

The repayment value increases at the end of the first year and at the end of each subsequent period of three months.

The repayment value *during* the first year is the purchase price only.

How much can you invest?

An individual may hold up to £5,000 (200 units) of 25th Issue. This is in addition to all other holdings of National Savings Certificates.

Denominations

Certificates can be bought in denominations of 1, 2, 4, 10, 20, 40 and 80 units of £25 each.

column headings and the data field; the other is the extra matter, a term used to cover all other information. The questions below looked at these two areas separately, and also addressed the issues of how the table was related to the text:

- Relationship of table to text
  1. What is the orientation of the table in relation to the body matter of the document?
  2. Where is the table located in relation to its textual reference (if any)?
  3. How is the table separated from the body matter of the document?
  4. Is the table differentiated from text (e.g. by background tint, different type size etc.)?
- Extra matter (top and bottom)
  5. How is extra matter aligned with the table body?
  6. How is extra matter separated from table body?
- Table body
  7. How are column headings separated from row headings or data items?
  8. Are rows grouped and separated in any way (e.g. they may be grouped by sense, or divided into groups of 3 or 5 and separated by extra line space or by a rule)?
  9. How are columns separated from each other (e.g. column 1 may be separated from the rest by a different means)?
  10. How are column headings orientated in relation to the rest of the table?
  11. How are row headings aligned with their column heading (if any) and with each other?
  12. How are data items aligned with column headings and with each other?
  13. How are data items aligned with row headings?
  14. How are successive lines of heading (if any) aligned with the first line?
  15. How are coded items signalled?

Figure 9: (Reproduced by permission of The Royal Bank of Scotland PLC)

TABLE 1

Number of persons covered by different health schemes in Japan (March 1980)

Scheme	*Employees (thousands)*	*Family dependants (thousands)*	*Others (thousands)*	*Total Number (thousands)*	*Per cent*
Employees' Health Insurance					
1. Government-managed	14,251	16,351		30,602	26·4
2. Society-managed	11,193	15,835		27,028	23·3
Day-Labourers' Health Insurance	333	209		542	0·5
Seamen's Insurance	216	461		677	0·6
National Public Service Mutual Aid Associations	1,194	1,870		3,064	2·6
Local Public Service Mutual Aid Associations	2,851	3,920		6,771	5·8
Public Corporation Employees' Mutual Aid Associations	813	1,299		2,112	1·8
Private School Teachers and Employees' Mutual Aid Association	312	272		584	0·5
National Health Insurance			44,552	44,552	38·4
Total	31,163	40,217	44,552	115,932	100·0

Figure 10: (Reproduced by permission of The World Bank)

Table 4a: Current-Account Financing of Developing Countries, 1978–81
(US$ billions.)

	1978	1979	1980	1981[a]
Current-account balance (excluding official transfers)	−45.4	−41.8	−72.8	−109.7
Official transfers	6.3	10.1	12.0	12.9
Private direct foreign investment, net flows	6.5	8.1	11.2	14.3
Medium-term and long-term loans, net disbursements	52.6	54.4	55.3	68.0
Official	13.6	15.1	20.6	24.1
Private	39.0	39.3	34.7	43.9
Other capital, net flows[b]	−10.2	−10.1	2.2	5.6
Change in reserves[c]	−9.8	−20.7	−7.9	8.9

Paragraphs appeared to a greater or lesser extent in all journals and leaflets. Tables, however, only appeared in journals. This means that any generalizations that can be made about the conventions of structure and realizations of tables apply only to journals and not between journals and leaflets. Examples of some of the tables used in the survey, which have the features described below, are shown in Figures 9–12.

Figure 11: (Reproduced by permission of the Commonwealth Institute of Entomology)

TABLE III. *Numbers and percentage survival to the pupal stage of* A. merus *larvae reared in various concentrations of sea water*

% sea water	No. of larvae tested: Nov. 1979 (48) I	Nov. 1979 (48) II	Jun. 1980 (52) I	Jun. 1980 (52) II	Sept. 1980 (67) I	Sept. 1980 (67) II	Total	% survival to pupal stage
0	25	25	16	10	21	22	119	79·3
20	14	9	17	18	18	22	98	65·3
40	25	16	21	18	22	23	125	83·3
60	13	19	15	20	22	21	110	73·3
80	0	0	19	11	12	10	52	34·7
100	0	0	20	17	0	0	37	24·7

25 larvae were used in each replicate at each dilution.
Figures in brackets indicate the number of *A. merus* egg-batches used.

The complexity of tabular matter resides in the number and kind of relationships which exist between its numerous units of information, all of which have to be signalled to the user. It is not surprising, therefore, that answers to the questions did not provide such a clear cut picture of the use of conventions as did those about paragraphs.

Tables were separated from the text by additional line space. In some cases the separation was further signalled by the use of rules or of a background tint. Extra large tables were placed on a separate page and were often rotated through 90 degrees to fit them in.

All the tables were made up of two parts which were called the extra matter and the table body. The table body covered row and column headings and the data field; everything else came under the heading of extra matter. The extra matter always contained identifiers, but other units such as table-related matter and keys to coded information were only present in certain circumstances. Identifiers tended to be located above the table body, except when they were the only item of extra matter, when they might be underneath the table. Keys to coded information and table-related matter were located beneath the table. Extra matter, whether above or beneath the table body, was separated from it by additional line space and/or rules.

Identifiers consisted of two smaller units of information, the table number and the table heading. Even if the two smaller units were placed on separate lines they were treated spatially as one unit. Thus if the table number was centred over the full table measure, the title would be as well. Whereas if the number was ranged left, the title would also be ranged left. If number and title were not on separate lines, they would be separated either by a

Figure 12: (Reproduced by permission of Academic Press, New York)

Table 2. Lead concentration (μg/g dry weight) in modern ribs

Concentration	*n*	Country	Reference
35	173	U.S.A.	Nusbaum *et al.*, (1965)
17	34	U.S.A.	Schroeder & Tipton (1968)
54[a]	60	U.K.	Crawford & Crawford (1969)
93[b]	46	U.K.	*Ibid.*
24[a]	27	U.K.	Crawford & Clayton (1973)
34[b]	28	U.K.	*Ibid.*
15[c]	16	U.S.A.	Holtzman *et al.* (1970)
10[d]	16	U.S.A.	*Ibid.*
14	5	U.K.	Hislop *et al.* (1973)
28[c]	21	U K.	Barry (1973)
17[d]	14	U.K.	*Ibid.*
14	45	U.S.A.	Gross *et al.* (1975)

[a]Hard water districts; [b]Soft water districts; [c]males only; [d]females only.

full point or a colon coming immediately after the number. Table number and title might be differentiated from each other by a different type variant or a different alphabet. Table number and title might be in a different type variant and/or type size from text type.

Table-related matter was usually set to the full text measure. Keys to coded information might be set to the full text measure as continuous prose, or be organised as list structures. Both might be in the same type size and variant as the table title, or as the table body.

Columns gave the table body its characteristic structure. The first column consisted of row headings which might have their own column heading. Successive columns consisted of a column heading and data items. In some tables data items might be totals of a row or a column.

Columns were separated from each other by space. Column headings were separated from the data field by additional line space and/or a rule. Where there was more than one level of heading, the superordinate headings were centred over the subordinate headings to which they related. Rules or braces, rotated through 90 degrees, might be used to show which subordinate headings belonged to the highest level heading. Headings at one level which ran on to more than one line had the lines centred beneath each other. The top or bottom line might align with single-line headings (called hung and stacked respectively).

Row headings ranged left unless there was more than one level of heading. If this was the case subordinate levels might be indented or, exceptionally,

might be in different columns. Row and column headings were in a smaller type size than text and might be in a different variant. Different levels of heading were in the same type size (though an italic variant might be used for a foreign language).

Data items were horizontally aligned with row headings. Data items which were column totals might be separated from other items in the column by a short rule above and one or two below. Where all data items in a row were totals a single rule might be used to separate them. When a column contained all totals no separation other than space was used.

Data items might be visually centred under each other, ranged left, ranged right or aligned around a decimal point. Data items might be alphabetical or numerical and might vary considerably in length.

The above descriptions show some of the variations that can occur in tables. At each point where decisions have to be made about how to signal changes in the text, to the reader, the number of choices open to the designer may be restricted. However, the permutations of choices are seemingly endless and account for the enormous variety in the appearance of tables.

7 Conclusions

The work begun in the British Library project shows the range of choices open to designers to signal and cue the reader to what is happening in the text. It also shows the kinds of restriction placed on those choices in particular circumstances, and the way in which choices made at one point, say in relation to the main text, relate to those at other points. Analyzing and describing text in this way is a slow process, but it is only by looking across a whole range of documents that the full range of possibilities both in structure and in realizations begins to emerge.

The author would like to acknowledge the permission of the British Library to reproduce Figures 1, 2, 3 and 4.

The Contribution of Linguistic Knowledge to the Interpretation of Text Structures

Jacques Virbel

Langages et Systèmes Informatiques
Université Paul Sabatier
118 route de Narbonne
31062 Toulouse Cedex, France

ABSTRACT:
In this article, we examine a series of assumptions allowing an approach, based on linguistic considerations to the problem of an abstract representation of text structures, with a view to handling them at the level of the physical production of text.

Taking as starting point the idea that it is possible to distinguish a specifically linguistic aspect from amongst all the aspects of the editorial process, we show that all the physical characteristics of the text (to which we shall refer as *physical page layout* correspond to undeveloped or only partially developed discursive traces, allowing their meaning to be studied within the language itself, and not within visual semiology alone. We show that these extended language forms can be regarded as speech acts concerning the text itself (as opposed to its contents or subject matter), leading us to the hypothesis that they have a meta-discursive value. We then briefly describe a syntactic method to distinguish and characterize the speech acts considered.

1 Introduction

In so far as a text is clearly a product of linguistic activity, and linguistics is defined as the science of such activity and of the languages allowing it to be carried out, it would seem a priori that the contribution of linguistics to the study of document structures should open up a very wide field of possibilities, since in the various aspects of language that it studies (syntax, semantics, pragmatics) linguistics examines a very great number of phenomena likely to appear in texts; moreover—and this is without taking account of the oral expression of language and *vocal documents*—certain areas of linguistics or neighbouring disciplines are, today, specifically concerned with texts.

We can link this preliminary observation with another: in fields of research and development specific to computer science for which *natural texts* are a vital concern, there is an increasing desire for the systems to be enhanced with elaborate and detailed knowledge in order to make them more reactive, autonomous, consistent or even *intelligent*; it should be noted that there are many such fields, including such obvious cases

are documentation and retrieval systems, office information systems, text editing and formatting systems, high level database management systems, *friendly* interfaces, etc. In the same way, alongside linguistics in the narrow sense, there are many fields of work inspired by linguistics, or which are developing on the basis of different approaches towards language, including, for example rhetoric, the study of argument and more generally of natural logic, the didactics of writing and writing environments, or again, at another level, psychology, memory, reading and editing processes, etc.

In addition, there has been the recent development of *computational linguistics* which tends to overlap with one of the major areas of artificial intelligence, *natural language understanding*, and also with areas of research associated with the increased use of the computer in society and as a means of communication—so-called *human factors* and ergonomics, for example.

Nevertheless, this apparently favourable conjunction of circumstances has to be viewed against a very varied background, and presenting these possible contributions from linguistics to the study of text structures gives rise to several orders of difficulty which it would be dangerous to ignore or to minimize.

1. In the first place, at the most general level, is the question of relations between computer science and linguistics. These relations are complex by nature, since for computer science, linguistics can be regarded as a field of application, a source of data, knowledge or of formalisms, a research field, or again an equal partner in the development of *cognitive science*, or all of these simultaneously. Without a doubt, the perception of computer science from the point of view of linguistics would be broadly symmetrical. These relations already have a long history, by no means free of difficulties, which started more than 30 years ago, above all in the context of the development of automatic translation and documentation systems, and it is clear that this history has contributed to the intellectual moulding of expectations and requirements.
2. Nor can we really speak of linguistics today without mentioning the main tendencies, schools and vocations it covers; and questions as apparently fundamental as the nature of language and the objectives of a science of language, its nature (an observational science or a formal science) the component elements of the language, the separation between levels, and when we turn to methods of analysis, or the na-

ture of appropriate formalisms and models, it is difficult to see any convergence in the main linguistic theories active today.

On this issue, as on the issue of the relations between computer science and linguistics, the development of computational linguistics, as we have already pointed out, is tending to create new fields of enquiry, or at least is leading to more or less fundamental changes in the methods and even the very objects of linguistic description and modelling.

3. When we come to texts, of the many questions over which the different linguistic approaches tend to diverge, none is perhaps so significant as the issue of the varied legitimacy of text (or discourse) as an object of study, and consequently that of the nature of texts (as opposed, for example, to sentences).

   The opinions of the various schools cover just about the whole range of possible views; for example to mention only the most clearly distinct:

   - the only object that linguistics can study is a minimal statement—a sentence—which is an entity of the language discourse is a (personal, social) implementation of the language—speech—and should be left to other disciplines.
   - an extended discourse includes units whose relations are homologous to those of the components of the sentence, which justifies the concept of a *text grammar* whose formal structure can be of the same type as that of sentence grammars.
   - Objects observable to linguistics are simply the statements actually produced—discourses—which linguistics attempts to break down into constituent elements—some of which will be sentences—using its own methods the sentence/discourse distinction may be nothing more than a stylistic variant.
   - A discourse is mainly a sequence of sentences the representation of a discourse is therefore simply the sum of the representations of the sentences that form it.
   - A discourse is a language object which is a legitimate area of study for linguistic research but it is a global reference level, and includes specific characteristics, requiring description and representation tools that have themselves to be original.
   - Linguistics is best suited to building up models of sentences, and can then collaborate in a multidisciplinary way with studies al-

lowing discourse to be perceived as a psychological, social, ethnological, historic, aesthetic, or other object.

This comparison of views shows that textual linguistics has developed, that it is varied, that its roots are in the identification and description of intersentence phenomena (of which dialogue and the question-answer situation are particular cases), and the integration of intra sentence observation within more extensive units, extending to all the properties of *textuality* (which means that a text is not the same thing as a random sequence of sentences): cohesion, consistency, anaphora and reference, thematization, connectors, etc., with the aim of building up models of types of texts and statements (narrative, description, argument, instruction, etc.).

Here, too, historic or anthropologic preoccupations also play a role, tending to give rise to a perception of a text through abstract cultural models (texts as megasentences, as a monument to visit, as a mechanism or organism, as an event, etc.).

This research work is becoming increasingly linked with various approaches developed within artificial intelligence (text generation, comprehension of stories, narratives, etc.) or in cognitive psychology and logic, and is becoming concerned with varied fields of applications: computer aided writing, didactics of writing, man-system communication, etc.

4. Finally, insofar as text is a **written** statement or discourse, there is the question of **writing** in linguistics. Historically, almost all structuralist tendencies, or those developing from them, show a degree of indifference, or even of outright exclusion, towards writing. Saussure's position on written language—like his position on the synchronous/diachronous dichotomy—is undoubtedly the source of such exclusion. Language for him was above all an oral phenomenon—which is undeniable—and writing systems were only a secondary coding process—a far more debatable proposition. It is possible, however, that Saussure's position has been forced into the interpretations of it furthermore, the fact that research explicitly inspired by him has tended to concentrate on phonology, and the syntax of sentences—and also the fact that American structuralism has been concerned with the description of oral languages—had masked the specific and more or less autonomous phenomena of writing systems, and consequently of those that appear in the space distribution of written text.

> Some research has however been concerned with writing systems, in particular based on didactic, historical or ethnological concerns, for example in the study of spelling, punctuation, textual bibliography, typography and reading. Less attention is being paid, on the other hand, as far as we know, to phenomena associated with the concrete makeup of a page or poster.

Sketchy though these comments may be, they show why we have felt a complete overview of questions was impossible within the scope of this article. We have attempted to make good this lack in part by including a bibliography which considers the most important of these issues and many of the references we give are themselves bibliographical sources on more specific points.

Our aim here will rather be to consider a particular but, we believe, central example: interpretation and linguistic analysis on the assumption that the structural phenomena of text have particular modes of graphic expression. This aspect has been selected in view of its importance for document structures, and also, we hope, because of its value as an illustration of the possible contributions a linguistic approach can make to this question.

2 A Linguistic Interpretation of Displayed Text Structures

A convenient, though oversimplified, way of setting the sources of this question within their context, is to compare it with the problem of formatting: can an abstract characterization—and of what type?—be associated with the elements of a document, in such a way as to be able to generate the formatting from the characterization, or to reformat the document automatically (e.g. in a way similar to that allowed by Scribe); or again we could consider the problem of editing control: what structural knowledge must a program have in order to be able to assist the text revision process? This question leads us to ask whether the *elements of physical page layout* of a document have a meaning in themselves, or whether they can contribute to meaning, or whether they are simply an addition—a mere *typographic enhancement*—downstream of the more fundamental core.

In general terms, the visual appearance of a text is the result of complex choices which are faced on the optimization of a set of varied constraints: the need to take into consideration certain aspects of the contents of a text, psychology of perception, economic and social conditions in publishing and

distribution (e.g. the editorial format and style imposed by certain series). These constraints form sets each of which can be associated with a particular point of view within the text. We shall show how this observation leads to our developing the concept of a *textual actor*, which in turn guided our research towards the methodology of multi-agent planning. The physical page layout of a supposedly *single* text can therefore vary considerably from one edition (version) to another, if the value of just one parameter is modified, and if the consequences of the dependencies between parameters are systematically applied. An examination of these variations, by observing existing corpuses or by direct experiment, shows that some of them, because they constitute interpretations or positions with respect to several aspects of meaning, even only indirectly or implicitly, can produce greater or lesser effects.

So, to take the example of headings, if the hierarchical structure of various parts of a text is only shown by physical properties of title segments, i.e. by physical page layout, (generally, the higher a section of text in this hierarchy, the larger the characters used for its title, though other conventions can be adopted), it is clear that modifications made to these properties can induce a different perception of the structure, and possibly modify the global meaning of the text. In the same way, if the internal structure of an enumerated list is shown by a set of diacritic symbols or indentation, the slightest modification will lead to a different form of enumeration, which may be very different in meaning from the first, since the nesting of levels within an enumeration often imply relations of inclusion, specification, implication, etc. At another level, we have all met the problem, in certain books, of inaccurate use of different types of quotation marks for quotation, stressing, indirectly reported speech, irony or polemics, etc; or again, that of nested or overlapping enumerations, some of which, although it is clear that they are aimed at providing the advantage of internal references independent of any page monitoring system, nonetheless make immediate memorization difficult, and can even make the text less comprehensible.

But, at the same time, we observe that other variations have no perceptible consequences on the meaning of the text in question, and that many forms of physical page layout can apparently be taken as synonymous.

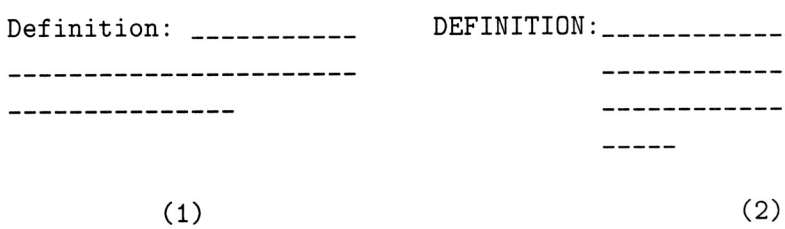
Basing ourselves on a large range of physical page layout phenomena, and generalizing from the implications to which they lead, we can reach the following conclusions:

1. The identification and characterization of certain types of textual

units can be based on methods inspired by physical page layout concepts.

2. A particular relation between identification and characterization, which can be called a function, implemented by physical page layout, can be introduced by a variable number of physical page layout processes (the number depends, as an absolute value, on the resources associated with a given physical device—typewriting, typesetting, or other). For example, a quotation can be distinguished by quotation marks (of various types), italics, a different font size from that of the current text, small capitals, a line feed, underlining, etc.; clearly, some of these processes can be combined with others.
3. The latter observation can be generalized: physical page layout elements are often redundant or cumulative. In fully general terms, this phenomenon is not always easy to interpret; depending on whether it produces an effect that is similar to stressing or highlighting, or, on the contrary, is the result of actual accumulation, by means of superimposition, of several different structural effects.

   For example, the difference between presentations (1) and (2):



   can be interpreted as a simple difference of degree in the contrast provided by the way a *definition* segment is produced, or as the combination of a contrast effect of this kind with another effect associated with ease in later consultation.
4. Flowing on systematically from point 2 above, we also observe that a single physical page layout mark can be used in the notation of very different functions. For example, italics can be used for quotation, allusion, stress, references, a conventional and regulated use, or more simply to provide for contrast between textual units of different types.
5. These characteristics are undoubtedly attributable to the fact that in general there are no explicitly shared conventions for their use, except, for some of them, in typographic tradition and art, or in restricted areas (*instructions to authors* for specialist journals and conferences,

book series formats, small ads, etc.). This means that conventions are created locally (in the limiting case, text by text) and that, consequently, it is much more the differential and contrastive values of elements in physical page layout, rather than their absolute values, which denote textual values and ranges. For example, it is rarely recommended that to denote a text segment as quotation, a specific font size should be used. On the other hand, it is quite common to use a font size contrast, reducing or increasing the size of characters used in a quoted segment by one or two points with respect to the current size, or even by a given proportion of the current size, within limits justified by other considerations (readability, line measure, etc.).

Our discussion above has highlighted certain features of physical page layout:

- a means of identifying and characterizing particular textual units;
- many-one correspondence between textual unit types and physical page layout unit types;
- in general, there is no absolute conventionality of the unit types, but differential and contextual values are used.

It is these features that we shall use for our characterization work, on the basis of a linguistic and logico-linguistic approach.

To do so, we base ourselves on a central observation, from which we deduce several interdependent consequences.

If we consider in detail the nature of the characterization operations carried out, we find that the multiplicity of physical page layout realizations for a single function corresponds to another multiplicity: a given physical page layout refers to a family of entirely or partially discursive counterparts.

Let us consider an illustration of this point, e.g. that of definition. At one extreme, we have a wholly discursive formula, such as:

1. *X is defined as Y* (where X and Y are any textual segments), and expressions linked to them by normalization, associable with support verbs or other transformations;
2. *(it is given + we assume + let + etc.) the definition (of X as Y + following: X is Y + etc.)*
3. *X (is defined + is definable) (by + as + as follows: + etc.)Y*
4. *Y (constitutes + represents + is + etc.) the definition of X*

At the other extreme, we have a formula of the type *X:Y*, where the segments X and Y will be realized using one of their various physical page layouts showing that the context is that of definition. An examination of

intermediate approaches suggests that there is a continuum between a formulation in which discursive expressive methods predominate, while expressive methods from physical page layout are more secondary, and formulations in which the relationship is reversed. For example, we might see:

1. an entirely discursive formulation (1 to 4 above)
2. a formulation with lexical separation associated with a particular page layout (character type, distribution through space); the extent of separation can vary, depending on the nature of the textual segments separated, and can even make use of reductions such as abbreviation, acronyms, etc.

   ```
   DEFINITION of X : X is Y
   DEFINITION of X : Y
   DEFINITION : X is Y
   DEF : X is Y
   ```

3. a formulation in which a lexeme from family *define* and its derivatives is no longer present, but where the act of *defining* is denoted by means of a physical page layout property which may correspond to:
   - the types of characters used (italics, bold, a particular colour, etc.)
     e.g.: X is *Y* (with Y in italics)
   - symbols, whether diacritic or other
     * X is Y
     # X is Y
     etc.
   - graphic symbols:
     || X is Y ||
     | X is Y |
     etc.
   - specific positioning:

     ```
                      ------------------------
                      ------------------------
     X is Y
                      ------------------------
                      ------------------------
     ```

In the same way, when dealing with an enumeration, there may be a variation in realization such as:

```
* A              1 -- A              1 -- A
* B              2 -- B              2 -- B
  -- C                 a -C                2.1 -C
  -- D                 b -D                2.2 -D
* E              3 -- E              3 -- E
```

(where A ... E are textual units of any length or structure)

- *(1) A; (2) B; (2.1) C; (2.2) D; (3) E.*
- *(To start with + first of all + at the outset + etc.) A; (then + secondly + to continue + in consequence of which + etc.) B; (part of which is + including + etc.) C; (and on the other hand + as well as + etc.) D; (and finally + to conclude + to terminate + etc.) + finally + etc.) E.*

Note again that in general conventions in these usages vary considerably, running from recognized usage through to specified ad hoc conventions (definitions tend towards the latter case, cf. dictionaries, encyclopaedias, technical or scientific manuals, etc.), and the methods used are combinable with each other to different degrees.

There are other remarks to be made about this example. For instance, a set of definitions included in text may be numbered, in various ways: first definition, 1st definition, definition no. 1, definition 1, etc. Furthermore, a textual unit may include one sentence, or a sequence of sentences, but may also be a phrastic element.

Analysis of this typical cases make it possible to establish that many physical page layout phenomena may be interpreted as non-discursive equivalents of certain statements that are developed and expressed on the basis of verbal expressions such as enumerate, title, define, quote, highlight, introduce, conclude, comment, refer to, make an observation, give an example, etc. We use these observations to postulate the following three correlated hypothesis:

1. The meaning of elements in the physical page layout, and their contribution to the meaning of the text, may be analysed within the natural language itself by an examination of their discursive counterparts (see section 2.1).
2. These discursive expressions evoke particular actions; we shall refer to them as particular speech acts, with a specifically textual vocation (section 2.2).
3. The effect of the meaning of these expressions can be regarded as

metatextual, with respect to the primary context of the text (its theme or subject, the objects or aspects of the world it handles) (section 2.3).

Combining these hyphotheses leads to the suggestion that there exists an *extended text* (or prototext) underlying any given text, and the perceptible state of this text (with its physical page layout) is a result of a series of deletions or reductions with respect to the prototext, operations which leave particular traces, in terms of a physical page layout (section 2.4).

Finally, we shall see that these hypotheses lead to what we believe is a new way of posing the problem of recognizing structural intentions on the basis of their expression in terms of physical page layout (section 2.5).

2.1 Analysis of physical page layout within language

This idea flows fairly naturally from the observations we are making above, and we shall not devote much attention to it here. It has, however, to be said on the one hand that this concept provides the basis in principle with the possibility of a semantic description as extensive and detailed as necessary for analysis of the contents of a sublanguage of a given language, formed by the set of discursive counterparts in a physical page layout, the alternative being the obligation to use only a physical description of the concrete realizations of the physical page layout, however subtle it may be (*semiology* of typography, *structural analysis* of layout of pages or posters, etc.). On the other hand, this approach allows the introduction, at least potentially, of the declarative or pragmatic part of this sub-language. For example, nothing prevents us determining different textual agents each of which uses one (or one type of) particular sublanguage (author, editor, publisher, printer, librarian, distributor, etc.), with the possibility of treating the superimposed or linked effects of these multiple sub-languages on a single text as a particular case of multilingualism.

A consequence of this approach is that it allows complete inventories to be produced of the terms or formulas of a language which can denote the elements of structural relations within a text, and to build up classifications of them based on linguistic properties. In section 3, we describe the basic elements of such a method.

2.2 The elements of physical page layout as traces of particular speech acts

An examination of the different sentence structures that can be used within the physical page layout, and of their conditions of use, suggest that they have the value of an act. For example, whether I write *I shall introduce my subject by saying A* or *INTRODUCTION : A*, I am asking my reader

to treat textual segment *A* as an *introduction* to what follows. In a sense, consequently, the status of A as an *introduction* depends on a particular act that I carry out, by pointing out how *A* is to be heard (understood, read, received, etc.). This *introduction* may, moreover, succeed or fail from the point of view of the art of introducing a subject but it draws its status as introduction from the effects of a particular textual act, and not, for example, from its conformity to a pre-existing model. In the same way, a *chapter*, a *section* or a *paragraph*, however *natural* and justifiable the motives advanced for their existence in a given text may be, owe their existence in the first place to the fact that in some way or another I perform the existence of such and such a textual entity with a particular status within my text.

When I mention this performative character, I refer to the theory of speech acts as it has been developed in Austin's initial work, and later by Searle, Strawson, etc. within the framework of analytic philosophy of language.

In a nutshell, we need simply say here, for the purposes of this article, that the theory of speech acts is based on the observation that alongside its affirmative role, based on the criterion of truth or falsehood, language is also the performance of acts by virtue simply of its use. For example, in *I promise to come*, *I declare the session open*, *I declare you man and wife*, the events promise, opening of session and marriage take place by virtue of the fact that a certain speech act constitutes them as such. The criterion here becomes that of success or failure, as in logic of action. Of the very many consequences to which this approach to the analysis of language has given rise, we shall simply consider two, that are relevant to our approach:

1. Precisely three fundamental types of speech acts can be distinguished, respectively called:
   - locutionary acts: this is the fact that a sequence of sounds belonging to a language is produced (or, in the case of writing, a particular sequence of graphemes belonging to an alphabet)
   - illocutionary acts: this is the act produced by the fact that a given loctionary act has been carried out; for example, *I promise to come* performs the illocutionary act *promise*
   - perlocutionary: this is the act performed in the hearer, by the fact that a given illocutionary act is carried out, for example, solicitude or threatening, e.g. in the case of a promise. The correspondence, or lack of it, between illocutionary and perlocutionary acts expresses the success or failure of the illocutionary act,

which therefore depends in part on the (linguistic) means used to perform it.

2. A second aspect concerns the direct or indirect nature of the illocutionary act, which is linked, as we shall see, with its conventional aspect. The issue here is to explain how illocutionary acts can be performed by non-explicit means. For example, *I shall come*, in which it is reasonable to assume that the explicit illocutionary formula *I promise that* has been deleted, and that explicit formula is a performed act; again, *could you shut the window?* or *it is cold here*, can be equivalent ways of requesting that a window be closed, although formally in the first case what we have is a question (concerning the ability of the listener to carry out the physical act which is the argument of the request) and in the second case an observation (on the state of affairs flowing from a failure to perform the physical act which is the argument of the request).

All the verbal expressions mentioned above as discursive counterparts of physical page layout may be analysed in such a way, as suggested by the example of *introduce* above.

Even statements whose conditions of uttering seem to be very remote from performance, such as theorems or demonstrations, in our view show this textual performance. For example, a theorem or demonstration, which can be shown to be badly formulated or to include an error nonetheless remains a *theorem* or *demonstration* in a given text if it is performed as such. In this case it would be a pseudo-theorem, or a false demonstration, in the universe of mathematics or of logic, but a theorem or a demonstration in that of textual performance.

In the same way, many aspects of punctuation are undoubtedly interpretable in the light of this assumption, including punctuation signs usually viewed as *syntactic* as opposed to so-called *expressing* signs. For example, considering sentences alone, although they lead to difficult problems of recognition in an oral context, the *fullstop blank capital* explicitly denotes where the writer expects the end of a sentence to be. Here again, such a unit may be regarded as unacceptable, ungrammatical or not in conformity with a particular standard, but that aspect of the unit has to be distinguished from the fact that it is performed as a sentence by its writer.

In general terms, textual performatives have an illocutionary influence oriented towards the text itself and show how the statements characterized by this illocutionary influence should be regarded as particular text units

within a text. It should be noted that they correspond very closely to class 5 of Austin's typology, *expositives*, or again, for example, to class 8T of Ballmer and Brennenstuhl's typology.

2.3 The meta-textual influence of textual performatives

In view of the fact that the information provided by textual performatives concern the status or type according to which the textual statement is to be considered, this information above all expresses a dimension which we can call meta-textual, with respect to the primary contents of the text. Here we are basing ourselves closely on Z. Harris' concepts, for whom language contains its own meta-language, and a discourse its own meta-discourse as a part of itself. Harris' analysis and above all his terminology, differs on these questions according to the different presentations that he gave to his theory, but essentially, it distinguishes two types of meta-linguistic operators:

1. those whose argument is the address of a segment of the discourse underway, or a quotation from this segment, e.g.:

   In the cat sleeps, *cat is a noun*
   In the cat sleeps, *the second word is a noun*

2. and those which do not contain such addresses but whose argument is a language entity, e.g.:

   Cat is a noun

These two types of operator give rise to sentences of the meta-linguistic type, which belong to the language and can be juxtaposed and coordinated with other sentences of a discourse. One of the properties of meta-sentences is that they can be reduced or deleted to a degree which varies considerably, in particular as a function of the context, including an extra-linguistic context, if that context is able to guarantee a certain form of coherence in the discourse, which Harris identifies as a repetition of words in successive sentences; these reductions and deletions take place leaving specific traces on the argument concerned, traces that can be recognized and interpreted in analysis.

For example, taking a very simple illustration, *are you coming?* (with a rising pitch) will be interpreted as a result of the reduction *I am asking whether you are coming*, where the deletion of *I am asking you* leaves a specific trace on its argument, a trace which is not a particular morpheme but a stress and a melody which affect the whole of the argument.

We can hardly go any further here in our discussion of Harris' positions. We hope, nonetheless, that we have shown that it is reasonably natural to

make use of this theoretical framework to interpret what we call textual performatives, since many of the procedures in physical page layout are indications of addresses of the beginning and end of certain textual units, or the characterization of certain operators whose meaning contributes to determining their status, as textual entities, within a given text. For example, in a physical page layout such as:

```
1.  CHAPTER I
    <title>
    A
```

there corresponds a sentence of the type

(1') *the text unit (which begins (at word i and ends at word j + at page i and ends at page j) + whose title is <title> + constituted by A + etc.) is the first chapter of this text.*

which is clearly meta-linguistic in the first two meanings mentioned above for Harris. The difference between (1') and (1) can be interpreted as a reduction resulting from various deletions and including traces of these deletions, here shown by the physical page layout.

2.4 Extended text (prototext)

A consequence of our meta-textual interpretation is to suggest that there exists an extended version of a given text, including all the meta-textual notations making the structure of the text explicit—this is a prototext to which one or several text versions may correspond, the physical page layout of each of these versions forming the trace (made up of a combination of more elementary traces) of reductions carried out on the prototext.

An idea of what a prototext might be can be obtained by imagining a world in which a text:

- would contain only one sentence, which could be as long as one wished,
- could only be written in a single type of character,
- could only be written on a single line—which could be as long as one required (imagine a horizontal roll).

In this fairly rarefied universe, what would be implemented, in other words, would be a minimal state of expressive resources, and the transfer from this prototext to an observable text would correspond to an intervention of richer expressive resources, whose role could be evaluated by observing their behaviour one by one, and then in their relations two by two and finally n by n. This would lead us gradually to the usage of segmentation

of text into sentences, and then to contrasts in characters, from a line to a poster by adding a new dimension, and then to a page with a further dimension, etc., until we reach the reductions mentioned above.

Such an object may appear arbitrarily abstract or remote from perceptible textual objects. It seems to us, however, that it is justified with respect to the assumptions we have made—at least as justified, in any case, as certain other syntactic objects not observed in the syntax of sentences, above all in the context of the reconstruction of elided or elliptical forms. It is, moreover, easy to imagine experiments allowing this reconstruction process to be controlled. Finally, it should be noted that certain textual objects show fairly clearly what the meta-linguistic role of a prototext could be, e.g. author summaries, analytic contents tables, or certain types of abstracts with a developed discursive form. From another point of view, our prototext resembles *Km records* used in certain word processing systems.

The questions that we shall consider below essentially, therefore, reduce to recognition of particular performatives within a text and characterization of the illocutionary effects of the textual units in question, that illocutionary influence being itself, in principle, interpretable, thanks to certain explicit locutory aspects, that can be identified with elements of the physical page layout or layout counterparts.

2.5 Intentions and conventions

It should be noted that in this approach, we encounter the concept of conventionality twice: once in the context of editorial usage concerning the physical page layout of texts—we have mentioned that these conventions are not absolute but have contrastive and local values—and again in the context of references to speech acts, and on their conventional aspects. Since speech acts, as Austin interpreted them, are conventional to the point where they could be paraphrased by an explicit performative formula, and if the means of expression in physical page layout, which are the counterparts of these formulas, are governed in their use by conventions based on editorial practice, the two concepts of convention can be superimposed to a greater or lesser extent. Since it is related to the question of characterizing illocutionary value or force, this formulation can be reduced to the establishment of a homologous link between editorial practice (e.g. typographic practice) and the conventionalism of speech acts (their capacity to be paraphrased by explicit performative formulas). This approach provides us with a means of investigation, which, moreover, is a context-sensitive tool, since, as we have already pointed out, the conventionalism of physical page layout is variable,

going from very common practices, some of which are apparently components of the very concept of a written text, to others, at the other extreme, which vary from genre to genre, from series to series, from author to author, and even from one text to the next.

This approach which is based on establishing a correspondence between the two forms of conventionality in the use of language can, moreover, draw on subsequent analysis based on various commentators on Austin, in particular, Strawson. The latter, who set out to examine the mechanisms allowing speech acts to be recognized without their being explicitly conventional, proposes that the core of this recognition, and its main instrument is the intentional of communicating meaning conventionally, using tools which make this recognition possible, but which, in consequence, do not necessarily have to be explicitly conventional, since their role can be limited to being a simple expression of the desire to communicate meaning conventionally, without being the explicit expression of the convention.

Examination shows that intentionality, in the sense that Strawson used the term (in particular starting from the *non-natural meaning* (Mnn) developed by Grice), plays the same role in textual performativity as in the general framework of recognition of the intention to express an illocutionary force and the recognition of the way (i.e. the means used) in which this recognition is expressed, if it is accepted that except in very particular cases (*automatic* texts of the surrealist type, *inspired* texts, or perhaps texts emerging from conditions of production of a type associated with an Orwellian universe), the author has every authority to speak about (and through) his text, i.e. as much authority as a referee has to decide on a penalty, a registrar to pronounce a couple man and wife, a chairman to open a session, etc. This idea, according to which an author not only has the intention of producing an effect in the reader, but also that of producing this effect by the fact that the reader recognizes it, and in addition the intention that this recognition should be one of the reasons for which the reader receives the effect, and that the recognition takes place in such a way that it can be recognized itself—makes it possible to understand how illocutionary intentions are recognized even when their intention is not (or is not systematically) conventional. A particular aspect, and one that is in the sense favourable to the role of physical page layout, is that the contrast principle (systematic identities or differences) can operate itself as a marker of intentionality, the illocutionary contents moreover being a local interpretation (to the text) or a more general interpretation when aspects of the physical page layout are clearly conventional in visual language.

3 A Method of Analysis

We showed in point A of the preceding section (here is a fine example of the beginning of a meta-textual sentence!) that an examination of the discursive counterparts of physical page layout can be carried out using linguistic methods to list and classify these elements. We have drawn up lists and classifications of this kind with the aim of forming a relatively general knowledge base, which can be used for various document handling programs (e.g. computer aided recognition, analysis, writing, revision or formatting).

Here we shall simply show very briefly how the assumptions discussed above can be turned to advantage at this level. One way of specifying and expressing these assumptions is to note that they can themselves be expressed in a discursive form. We can set out to build up sentence schemas expressing the relationship that we are attempting to establish between *candidate* performatives and usages of the physical page layout. Basically, the issue is to make a heuristic method out of the observation that the performative range of physical page layout can be analysed within the language.

The most general form of a textual meta-language sentence could be a formula of the type

$$N0_{at}\ V1\ [prep]\ N1_{at}\ \text{that}\ P$$

where

- Ni_{at} denote the nouns representing textual actors, in the sense in which we have already used the term (author, writer, revisor, typist, layout artist, typesetter, publisher, reader)
- Vi represents a family of verbs denoting the intention expressed by $N0_{at}$, a *textual* intention as we suggested before. E.g.:
  $V1$ = *express + translate + display + mark + underline + indicate + etc.*
- [*prep*] denotes a preposition or a prepositional phrase giving the meaning of the intentionality towards $N1_{at}$:
  [prep] = for the attention of + for + with + for the eyes of + etc.
- P is a proposition making the performative act itself explicit.

The main part of a sentence in this family consequently expresses the intentional aspects of textual performativity. Note on the one hand that the re-use of this part instead of P makes it possible to produce complex sentences expressing complex intentions (the author tells the typist that the typesetter should be told P); and on the other hand, that if any combination of Ni_{at} generates no interpretable or simply plausible sentence,

others will reasonably naturally express situations that are generally poorly or only partially integrated into representations of textual processes (e.g. with $N0_{at} = typesetter$ and $N1_{at} = author$, we simply express the function of proofing before printing). In the same way, it is easy to express the fact that the discursive part of physical page layout may have an author other than the author of the text (segmentation into various divisions of previous single-discourse texts in modern editions, written interviews with embedded interviewers' headings, etc.).

Let us return to $P\#$; the issue here is to build up sentence schemas expressing a performative textual act carried out by a textual actor by making use of the physical page layout in a certain way, and whose object is one or more textual units. At an abstract level, P will therefore have the following form:

$$N2_{at}\ Vm\ d\ N_{ut_i}$$

where

$N2_{at}$ again denotes a textual agent, which may be the same as No_{at} or $N1_{at}$

Vm represents a performative verb (or expression) with a counterpart in terms of physical page layout

d is a dependency link marker (preposition, conjunction, etc.) which depends on the syntax of Vm

N_{ut_i} represents one or more textual units providing the basis for the act represented by Vm, $i \geq 1$

Schemas for P would, for example, include:

$P1 : N2_{at}\ Vm\ N_{ut_1} \ldots N_{ut_n}$
e.g.: *(share + segment + divide + partition + etc.)* into *(chapter + section + parts + paragraphs + etc.)*
e.g.: *(number + count + list + enumerate + etc.)*

$P2 : N2_{at}\ Vm\ N_{ut_1}$ (by N_{ut_2})
e.g.: *entitle + introduce + conclude + preface + postface + etc.*

$P3 : N2_{at}\ V_m\ N_{ut_1}$ (and + to) N_{ut_2}
e.g.: *(chain + concatenate + etc.)*

$P4 : N2_{at}\ Vm$ that N_{ut_1}
e.g.: *(warn + advise + guarantee + observe + recommend + etc.)*

We give only a few examples of such schemas, some of which are undoubtedly debatable from the point of view of syntax. Note that several

hundred terms (verbs or nouns) will make such schemas valid, and that in addition an examination of other syntax properties is needed to show relevant facts in physical page layout terms, e.g. nominalization. Moreover, various types of coordination or complementation between such schemas, or schema elements have to be analyzed to account for the compositional aspects of textual structures.

The most general observation that we can make concerning this approach is that the use of such phrastic schemas, i.e. their construction followed by searching for the verbs that make them acceptable (acceptability here has both a syntactic and, as it were, *editorial* meaning), is certainly an invaluable means of producing complete lists, but does not provide a sufficiently powerful tool for description or classification given our aim (which is also *editorial*). For example, to take a simple case, nothing in the syntax of *divide* or *segment*—applied to a text—indicates that a chapter cannot itself, in principle, be divided into chapters, and a fortiori into volumes. It is clear here that the unacceptability P = *the author divides chapter 3 into 6 volumes* is a consequence of the nature of the direct and indirect objects of *divide* and therefore derives from a *law* of text construction and not of syntax. This is a fundamental problem in the interrelationship of the syntactic properties of language objects and of the properties, which may be physical, of objects in the universe handled by language objects. All we have set out to do is to show by means of a simple example, the form the general problem can take when the universe in question is that of publishing (i.e., from certain points of view, that of language). One direction for research to solve such problems is clearly *knowledge representation*, as it is used in artificial intelligence, where dependencies of this type, and other more complex dependencies, can easily be formalized, and modelling complex language activities, which for our part we can see no way of avoiding, not only in the new form of activity opened up by the existence of personal micro-computers, but still less in what is beginning to be called *industrialization of language.*[1]

[1]The thoughts described here are based on the THOT research programme in which we are engaged, within the Formaliziation of Reasoning team in the Computer Languages and Systems Laboratory of the University Paul Sabatier in Toulouse. Other researchers in this programme are M. Mojahid, E. Pascual and S. Tazi. We presented a list and classification for a first core of textual structuring functions in [Vir85, Vir86, TV85b, TV85a, Taz85]. A first approach towards aided editing was described in [Sop86], and another relating to planning of multi-agent formatting is described in [Moj86].

Performing Textual Structures in Documents

Saïd Tazi

UER Informatique, Université des Sciences Sociales
Place Anatole France
31070 Toulouse Cedex, France

1 Introduction

Textual structures form a field of study within natural language which can have valuable applications in document handling or computer-aided editing.

The basic idea here is to represent textual structures within a document through the activities of creation, revision and restitution. This can also be seen as a form of modelling for a certain type of reasoning about these different activities. The nature of the materials used in texts on the one hand and the methods used in interpreting linguistics on the other have led us to the following conclusions (which we shall examine in detail below):

- traditional and currently accepted models are insufficient or even inadequate to represent certain text structures.
- We have adopted a functional notation for text structuring actions interpreted by the use of a number of visual markers. This notation, called text structuring function (TSF), is used in document handling. TSFs provide a conceptual model of the TEXTLOG editing system which is currently being developed with some modules already implemented.

TSFs can also be used as a basic model for the design of a document formatting and restitution system.

2 Limits to current models

There are a great many textual phenomena whose representations do not correspond naturally to a hierarchical model [TV85a]. Amongst these phenomena, which may be involved in editorial manipulation (in the widest possible sense) for which a tree-structure is inadequate, there are the following:

Figure 1: The cone structure

```
Chapter I
definition 1: XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXX
definition 2: XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX

Chapter II
Definition 3: XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
```

- *cone* structures
- discontinuous level structures
- nested structures within a sentence.

2.1 Cone structures

We use the term *cone structures* for texts organized in such a way that the numbering of the various text objects takes place in parallel with one or several other numbering systems on objects that are shown sequentially.

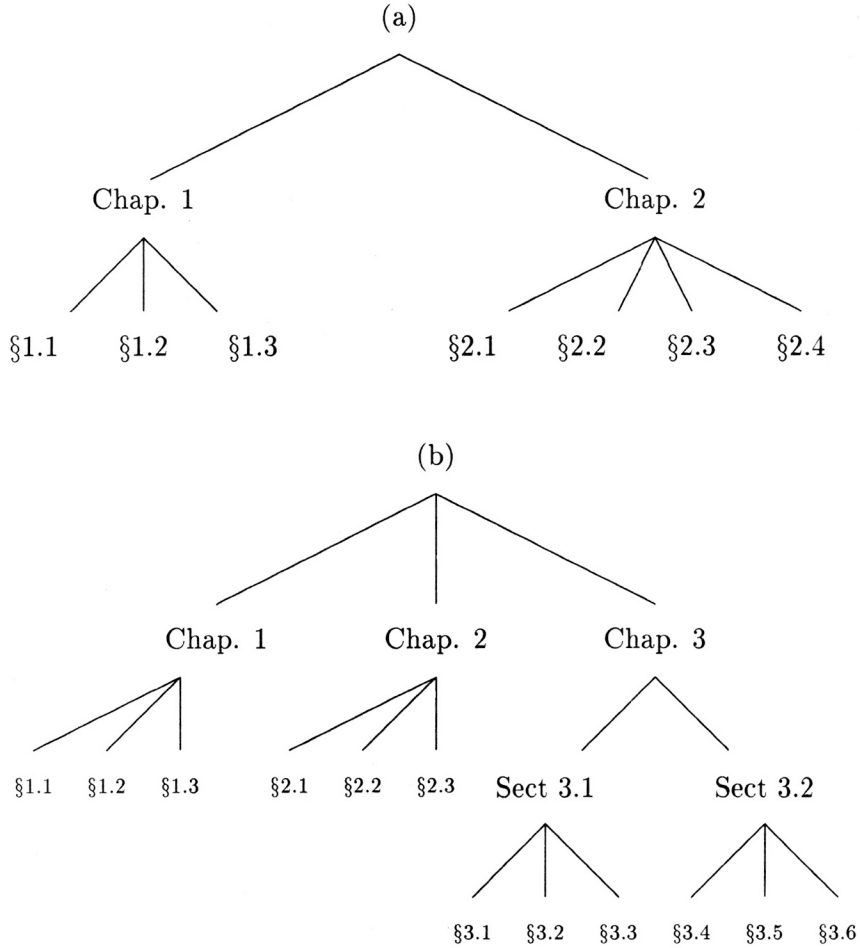
E.g.: consider the text in figure 1, where 'X' represents the text.

The tree-structured model seems inadequate to represent this type of numbering. The figure shows that *chapters* can be represented by the nodes of a tree, and *definitions* as subtrees to each *chapter* node; but the numbering system does not correspond to the continuity of the nodes (in depth or breadth) of each subtree. The *definition* textual units are not at the same level of the tree structure and breadth-first numbering cannot be used.

2.2 Discontinuous level structures

In the hierarchical representation of a document, each level is independent of the others. There are, however, phenomena in which there is no such independence. Figure 2 shows an example.

Figure 2: Discontinuous level structure



The articles in structure (a) are all at the same level and there is therefore a correspondence between the level of depth and the level of the lexical marker §. On the other hand, structure (b) does not present the articles at the same level of depth even if all the § nodes can be regarded as being of the same (logical) nature.

The problem posed by such a representation (the case of (b)) is that for any process applied to § (e.g. printing of § with a specific level of indent), we come up against the difficulty of establishing the appropriate hierarchical level (in depth); i.e. it is impossible for example, to associate articles with a special parameter relative to their hierarchical level.

There are other similar phenomena to the ones we have considered here, such as what we have called discontinuity in mark inheritance [Taz85].

2.3 Nested structures within sentences

Sentences are generally defined as a sequence of words, the first character of the first word being a capital, and the end being marked by a punctuation sign. We have, however, met the following textual structures which contradict this definition:

1. a sentence may contain an enumerated list, like the one you are reading now. It begins at "We have, however", and ends with the word "recognize". Note that there are other nested sentences within this first enumerative unit;
2. quotations or the use of parentheses allow sentences to be nested within a sentence (by way of illustration: The present sentence beginning with the words "The present" can be read as a sentence nested within the surrounding sentence), and which may end within the quotation or parentheses.

which make sentences ambiguous to recognize.

If we decide that the leaves of a tree which is the structure of the text are sentences, the problem that arises is that of deciding where nested sentences are to be located with respect to the sentence containing them.

3 Linguistics and textual structures

In most cases the various schools of linguistics have been exclusively concerned with the intraphrastic analysis of texts. Even those which studied relations between sentences within a text (discourse), from the stylistic point of view, have not—as far as we know—taken account of the positional dimension of text on a medium.

In practice, the message communicated by the author by means of a written text is naturally contained in the meaning of the sentences written in the form of character strings, but it is also communicated by the way in which that string is laid out. In other words, the layout of texts on a medium is a meta-textual component which provides a complementary means to assist the author to communicate his message.

All the physical characteristics involved in the production of a text on a material medium (such as the nature or the style of characters, layout, etc.) contributes in a specific way to the meaning of the text at several levels [SBB80]. This contribution does not belong to the character string

which forms the message, which leads to our saying that this contribution is non-discursive in the sense in which the term is used by Virbel [Vir82, Vir85].

A systematic analysis of all typical cases of the physical characteristics of texts allows us to establish that their semantics is such that they can be interpreted as non-discursive equivalents of certain textual actions such as *ennumerate, entitle, quote, introduce, stress, conclude, comment, refer, illustrate*, etc.

The textual actions mentioned above have a performative effect in the sense used in speech act theory [Aus70], i.e. semantics expressed by a performative act of speech lies in the enunciation of the act itself. In order to determine that part of a text is an introduction, the author performs it using the label *introduction* or by saying, for example, "I shall introduce my comments by saying...". For further details, see [Vir86] and [Vir88].

4 Textual structuring functions

4.1 Definitions

Two types of structures will be considered, the first corresponding to the linear mode of writing (the most common), and the second to non-linear ways of organizing writing (top-down or bottom-up). The notation that we have adopted to represent the first type of structure is that of linear lists, which have the expressive ability to represent the *chaining* of textual objects.

We use the term textual structuring functions for functions that represent the action of organizing a text (or part of a text) in a way defined by a certain structure. The TSFs take the form $F(M,T)$ where:

F is used for the name of the action which the author wishes (or wished)to apply to text T, using marker M;

M is the marker used to segment the text: this marker is distributed by concatenation over all textual segments which form T each textual segment then takes the logical status carried by the semantics of this marker;

T is the structured text, the list of textual units
$(T = [U_1, U_2, \ldots, U_n])$

There are many textual structuring functions. We have considered in detail those which make up enumeration: *list, number* and those which can be used for heading and segmenting text: *divide, enunciate, separate* and *entitle* [Taz85].

Figure 3:

```
xxxxxxxxxx
xxxxx:
*) xxxxxxx
xxxxxxx;        U1
*) xxxxxxx           T
xxxx;           U2
*) xxxxxxx
xxxxxx.         U3
```

Figure 4.

```
xxxxxxxxxxxxx
xxxx:
I xxxxxxxxxxx

II xxxxxxxxxx

III xxxxxxxxx
```

Figure 5.

```
xxxxxxxxxxxxx
xxxx:
CHAPTER I
xxxxxxxxxxxxx
CHAPTER II
xxxxxxxxxxxxx
CHAPTER III
xxxxxxxxxxxxx
```

4.2 Combining TSFs

In order to input complex textual structures, we have defined two ways of combining TSFs: combination + and combination *recursive*. We leave it to the interested reader to consult [Taz85] details on these combinations from. In general terms, combination + expresses the concatenation of different TSF markers within a single TSF; if a TSF has a marker M which can be broken down into other markers $M_1, \ldots, M_n$ corresponding to other simpler TSFs, we say that this TSF is obtained by combination + of the TSFs whose markers are $M_1, \ldots, M_n$. Recursive combination corresponds to the case where the textual units of a TSF are themselves formed from other TSFs, which corresponds to the nesting of textual structures within each other.

4.3 Examples

A. Consider the text represented in figure 3, where each row of xs represents a line and '*') is the marker used for the concept of numbering.

Figure 6:

This enumeration structure can be represented by the following text structuring function:

```
List ('*)', [U1, U2, U3])
```

The TSF *list* represents the enumerated text in the form of a list and is marked by `'*)'` The enumerative list `[U1, U2, U3]` represent the enumerated text.

B. To illustrate combination +, let us consider the following enumeration represented by TSF `number ('numeric', [U1, U2, U3])` in figure 4.

Figure 5 represents a combination + the enumeration in figure 4 and division into chapters, giving us the TSF:

```
Divide (CHAPTER, number ('roman caps', [U1, U2, U3]))
```

C. An example of recursive combination is shown in figure 6.

In figure 6, we have a numbering system marked by `1)`, `2)` and `3)`, whose second numbered unit includes a new enumeration using markers `a)`, `b)` and `c)`, representing the following recursive combination:

```
Number ('numerical'), [U1, [U20, number ('lowercase
        alphabet'), [U21, U22, U23])], U3])
```

The formalism of TSFs is a representation model for document textual structures. Some of the phenomena discussed in section 3 have been represented using TSFs, but we have not yet examined the power of TSFs to represent the phenomenon of nested sentences described in section 2.

Consequently, TSFs, which are, from the implementation point of view, interpreted as Prolog language clauses, are natural and at least sufficiently powerful to represent the phenomenon of nested structures. They form the conceptual model for textual structures in a text editing system called TEXTLOG which we have designed and some of whose modules, in particular the unparser, have been proven in an implementation. The formalism provides a general basic model for the design of systems concerned with natural text, such as restitution systems (formatting and typesetting), assisted editing systems, etc.

5 TEXTLOG

A document handling system has to be based on a conceptual model of the document. We defined the various functionalities of the TEXTLOG system on the basis of a textual structure representation using TSFs. In its present state, TEXTLOG is a system allowing editing of textual structures. These structures are recognized automatically using an analysis module and are then stored in the form of a TSF representation. The system is designed in such a way that textual structures can be modified in a controlled manner.

The text is input in the first place using natural editing conventions, within a host editor, and is then analysed, i.e. the markers used to perform the various structures are recognized. These markers are those that are used—with a minimal number of conventions—to write a manuscript that is to be typed by a secretary, such as carriage returns, the use of indentation or underlining. Editing operations are controlled by an automaton. Every time we are in logical textual object O, a pointer is in the corresponding state. Any attempt to modify the structure of a document is first checked at the level of the automaton. A modification to a textual object will be allowed or disallowed depending on whether there is a transition between the state corresponding to the object in question and another state corresponding to the objects obtained after modification. This transition must correspond to the type of modification required [Sop86].

Another part of this system is currently in design [Moj86]. It concerns document restitution on an output device, on the basis of its structure. The restitution of the document follows a philosophy similar to that of Scribe [Rei80a]. We have established an editorial knowledge base within which presentation schemas have been formalized for the various types of document. Each logical object of the document expressed in terms of TSFs is associated with a rule in the editorial knowledge base. Certain modules of the system have already been implemented, such as the analyser. The others are currently being designed.

6 Conclusion

We have shown, how we draw on the concepts of formal linguistics, *speech acts*, to represent textual structures in order to integrate them within a document handling system. The main result obtained at this level is highlighting of the correspondence between structural characteristics of texts, and a particular family of performative acts.

We have defined a new formalism within which the structures of a document can be represented: text structuring functions. These are then interpreted as clauses in the Prolog language, which we have used to implement the various modules. The TSF concept has been shown to be sufficiently powerful and natural for efficient handling of document structures. It can also be used for the design of systems which have to take account of text structures, such as a computer assisted aided system.

Finally, we have given a brief description of the TEXTLOG system which is currently being developed: this is an editing system based on the TSF concept, within which an analyser recognizes textual structures and a control module checks the consistency of structures before allowing revisions to be made.

By Way of a Colophon

This book is one of the first to be published in "The Cambridge Series on Electronic Publishing," from Cambridge University Press. The topic of the book is an aspect of that area, and further the details of its preparation and formatting provide a snapshot of the existing capabilities and potentials for electronic preparation and transmission of documents.

The book originated with a series of lectures given during an INRIA winter school held at Aussois, France, in January 1987. The draft documents written for that school were prepared with many different formatters, such as LaTeX, MacAuthor, Mint, Scribe, Word, or even by typewriter, and each was prepared in its own style. Furthermore, the language used in the papers reflected the differing languages of the lecturers—half of the papers were written in French and the rest in English.

Our work, as editors, began when we decided to merge the individual papers' bibliographical references into a single list. As possible, the individual authors' lists have not been retyped—the electronic form of the original lists were collected, converted into BibTeX, and merged together, in part using software we developed to allow selection of parts of BibTeX source files.[1] Transport of files was over networks such as the EARN, BITNET, the Internet (composed of the Arpanet, Milnet, and associated local area networks), and in large part over UUCP links.

Secondly, we translated French-language papers into English. The translations have been made by David Beeson, the translator of *TSI* (a French magazine dedicated on Computer Science published both in French by Dunod, and in English by John Wiley). Beeson delivered English transla-

[1] Further information about this work can be obtained from Richard Furuta.

tions prepared with Word on an IBM-PC. After the individual authors examined and updated their chapters, we converted the Word version into LaTeX, primarily by hand. The papers were then transferred over UUCP/Internet from INRIA to the University of Maryland. The chapters' bibliographic citations were converted from textual strings to LaTeX `\cite` commands at this time, and other cross-references were converted to symbolic form as well. Index terms were identified manually and were encoded into the LaTeX source files using the LaTeX `\index` command. Postprocessing necessary to form the index was performed by Scott Simpson's `index` program.

The figures have been prepared in a number of ways: LaTeX's `picture` mode, with Apple Macintosh products such as MacDraw and MacPaint, with the Xerox ViewPoint workstation, and others. Consequently, some of the figures have been stripped into the camera-ready form.

The document style used in this book is modified from that specified for the papers presented at EP88, the International Conference on Electronic Publishing, Document Manipulation and Typography [vV88]. All papers were formatted to that style and adjusted as needed at the University of Maryland, and then transferred back to INRIA. Then, the `DVI` file were processed on a Linotronic 300 typesetter by Imprimerie Louis Jean at Gap (France), using the TeX Computer Modern outline fonts redesigned with Metafont by Victor Ostromoukhov, to be compatible with PostScript.

Bibliography

[Ado85a] Adobe Systems Incorporated. *PostScript Language: Reference Manual.* Addison-Wesley, 1985.

[Ado85b] Adobe Systems Incorporated. *PostScript Language: Tutorial and Cookbook.* Addison-Wesley, 1985.

[AGQ85] Jacques André, Yann Gründt, and Vincent Quint. Towards an interactive math mode in TEX. In Dario Lucarella, editor, *Proceedings of the First European Conference on TEX for Scientific Documentation*, pages 79–92, Addison-Wesley, 1985.

[AHLM84] Robert M. Ayers, James J. Horning, Butler W. Lampson, and James G. Mitchell. *Interscript: A Proposal for a Standard for the Interchange of Editable Documents.* Technical Report, Xerox Corporation, Palo Alto, California, January 1984.

[AI84] Jacques André and Rolf Ingold. High quality book production using a text editor: A case study. In J. J. H. Miller, editor, *An Introduction to Text Processing Systems: Lecture Notes of a Workship held in association with PROTEXT I, the First International Conference on Text Processing Systems*, pages 13–20, Boole Press, October 1984.

[AM85] Vincenzo Ambriola and Carlo Montangero. Automatic generation of execution tools in a GANDALF environment. *The Journal of Systems and Software*, 5(2):155–171, May 1985.

[And87] J. André. Grif plus Mint, or how to abide by a layout-sheet. In J. J. H. Miller, editor, *PROTEXT III: Proceedings of the Third International Conference on Text Processing Systems*, pages 89–96, Boole Press, 1987.

[ANP81] Todd Allen, Robert Nix, and Alan Perlis. PEN: A hierarchical document editor. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, 16(6):74–81, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981.

[App84] Apple. *MacWrite.* Apple Computer, Inc., 1984.

[AQ87] Jacques André and Vincent Quint, editors. *Structures de/for documents*, January 1987. Support for an INRIA course, Aussois.

[AS85] Jacques André and Richard Southall. Experiments in teaching METAFONT. In Dario Lucarella, editor, *Proceedings of the First European Conference on TEX for Scientific Documentation*, pages 141–153, Addison-Wesley, 1985.

[Aus70] J. Austin. *Quand dire c'est faire (When Saying Is Doing).* Seuil, 1970.

[Bar65] Michael P. Barnett. *Computer Typesetting: Experiments and Prospects.* The MIT Press, 1965.

[Bau78] Patrick C. Baudelaire. Draw manual. In B. W. Lampson and E. A. Taft, editors, *Alto User's Handbook*, Computer Science Laboratory, Xerox Palo Alto Research Center, November 1978.

[BB81] Th. Ballmer and W. Brennenstuhl. *Speech Act Classification. A Study in the Lexical Analysis of English Speech Activity Verbs.* Springer Verlag, 1981.

[Bea85] Richard J. Beach. *Setting Tables and Illustrations with Style.* Ph.D. dissertation, University of Waterloo Computer Science Department, Waterloo, Ontario, Canada, May 1985. Also issued as University of Waterloo Computer Science Department Technical Report CS-85-45 (May 1985) and Xerox Palo Alto Research Center Technical Report CSL-85-3 (May 1985).

[Bea86] Richard J. Beach. Tabular typography. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 18–33, Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[BEF84] Ted J. Biggerstaff, D. Mac Endres, and Ira R. Forman. TABLE: Object oriented editing of complex structures. In

Proceedings—7th International Conference on Software Engineering, pages 334–345, IEEE Computer Society, 1984.

[BK84] Jon L. Bentley and Brian W. Kernighan. *GRAP—A Language for Typesetting Graphs; Tutorial and User Manual.* Computing Science Technical Report 114, AT&T Bell Laboratories, Murray Hill, N.J., December 1984.

[BK86] Jon L. Bentley and Brian W. Kernighan. GRAP—A language for typesetting graphs. *Communications of the ACM*, 29(8):782–792, August 1986.

[Bon85] P. R. Bono. A survey of graphics standards and their role in information interchange. *IEEE Computer*, 18(10):63–75, October 1985.

[BQ85] Bruno Borghi and Stéphane Querel. Vers un serveur d'impression sur sm90 (towards a printer server on sm90). *Actes des journées SM90 (Proceedings of the SM90 seminar)*, December 1985.

[BRV83] G. Bogo, H. Richy, and I. Vatton. Un modèle de représentation de documents généralisés. In *Actes des journées sur la manipulation de documents*, INRIA/IRISA, Rocquencourt, France, 1983.

[BS83] Richard Beach and Maureen Stone. Graphical style: Towards high quality illustrations. *SIGGRAPH '83 Conference Proceedings, Computer Graphics*, 17(3):127–135, July 1983.

[CCHM86] Peehong Chen, John Coker, Michael A. Harrison, Jeffrey W. McCarrel, and Steve Procter. The VORTEX document preparation environment. In Jacques Désarménien, editor, *TEX for Scientific Documentation*, pages 45–54, Springer-Verlag, 1986. Lecture Notes in Computer Science, Number 236.

[CCI86] CCITT. *Recommendation T73. Document interchange protocol for the telematics services.* CCITT, April 1986. CCIT-COM VIII-59.

[CD69] D. Crystal and D. Davy. *Investigating English Style.* Longman, London, 1969.

[CdVS86] D. D. Cowan and G. de V. Smit. Combining interactive document editing with batch document formatting. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 140–153, Cambridge University Press, April 1986. Pro-

ceedings of the international conference, University of Nottingham, 14–16 April 1986.

[CFB86] Ester Crisanti, Alberto Formigoni, and Paco La Bruna. EasyTEX: Towards interactive formulae input for scientific documents input with TEX. In Jacques Désarménien, editor, *TEX for Scientific Documentation*, pages 55–64, Springer-Verlag, 1986. Lecture Notes in Computer Science, Number 236.

[Che81] Lorinda Cherry. Computer aids for writers. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, 16(6):61–67, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981.

[CIV86] Giovanni Coray, Rolf Ingold, and Christine Vanoirbeek. Formatting structured documents: Batch versus interactive? In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 154–170, Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[CKST81] Donald C. Chamberlin, James C. King, Donald R. Slutz, Stephen J. P. Todd, and Bradford W. Wade. JANUS: An interactive system for document composition. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, 16(6):82–91, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981. This report was also issued as IBM Computer Science Research Report Number RJ3006 (37371), IBM Research Laboratory, San Jose, Calif., December 1980.

[CKST82] Donald C. Chamberlin, James C. King, Donald R. Slutz, Stephen J. P. Todd, and Bradford W. Wade. *JANUS: An interactive document formatter based on declarative tags.* IBM Computer Science Research Report RJ3366 (40402), IBM Research Laboratory, San Jose, Calif., January 1982.

[CLP81] Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier. The memory system of a high-performance personal computer. *IEEE Transactions on Computers*, C-30(10):715–733, October 1981. Also contained in [LCMO81].

[dBD81] R. A. de Beaugrande and W. Dressler. *Introduction to Text*

Linguistics. Longman, London, 1981.

[Dew86] M. E. Dewey. Tblp - a simple preprocessor for tables. *Software—Practice and Experience*, 16(8):731–738, August 1986.

[DKLM83] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Outline of a tool for document manipulation. In R. E. A. Mason, editor, *Proceedings of IFIP 83, Paris*, pages 615–620, Elsevier Science Publishers B.V., 1983.

[DKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Mélèse. Document structure and modularity in Mentor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Software Engineering Notes*, 9(3):141–148, May 1984. Also *SIGPLAN Notices* 19(3), May 1984.

[DLM84] V. Donzeau-Gouge, B. Lang, and B. Mélèse. Practical applications of a syntax directed program manipulation environment. In *Proceedings—7th International Conference on Software Engineering*, pages 346–354, IEEE Computer Society, 1984.

[DP86] L. Delamare and J. Paris. DOCUMENT: Un environnement du poste de travail Concerto. *T.S.I.*, 5(4):327–332, July–August 1986.

[DP87] L. Delamare and J. Paris. Document: An Environment for the Concerto Workstation. *T.S.I. (English edition)*, 6(1):89–94, 1987. John Wiley,publisher. Translation of [DP86].

[Dre86] M. Dreyfus. Mathor. *Micro-Bulletin*, (19):23–36, April 1986. Issued by C.N.R.S.

[dS65] F. de Saussure. *Cours de linguistique générale (General Linguistics).* Payot, Paris, 1965.

[EKMR83] J. Estublier, S. Krakowiak, J. Mossière, and Y. Rouzaud. Design principles of the Adèle programming environment. In *International Computing Symposium on Application Systems Development*, ACM, March 1983.

[ES85] Robert J. Ellison and Barbara J. Staudt. The evolution of the GANDALF system. *The Journal of Systems and Software*, 5(2):107–119, May 1985.

[FD82] James D. Foley and Andries Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982.

[FF62] G. L. Fischer and A. L. Fischer. *Optical Character Recognition.* Spartan Books, New York, 1962.

[FQA88] Richard Furuta, Vincent Quint, and Jacques André. Interactively editing structured documents. *Electronic Publishing—Origination, Dissemination, and Design*, 1(1), March 1988. To appear.

[FSS82] Richard Furuta, Jeffrey Scofield, and Alan Shaw. Document formatting systems: Survey, concepts, and issues. *ACM Computing Surveys*, 14(3):417–472, September 1982.

[Fur86a] Richard Furuta. An Integrated, but not Exact-Representation, Editor/Formatter. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 246–259, Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[Fur86b] Richard Furuta. *An Integrated, but not Exact-Representation, Editor/Formatter.* Ph.D. dissertation, University of Washington, Department of Computer Science, Seattle, WA, 1986. Also available as Technical Report No. 86-09-08, Department of Computer Science, University of Washington (August 1986).

[Fur87a] Richard Furuta. Structured document models and representations. In J. J. H. Miller, editor, *Issues in Generalized Text Processing: Lecture notes of a Short Course held in association with PROTEXT IV, the Fourth International Conference on Text Processing Systems*, pages 1–14, Boole Press, 1987.

[Fur87b] Richard Furuta. Trends in interactive preparation of documents. *Digest of Papers, Spring COMPCON 87*, February 1987.

[Fur87c] Richard Furuta. Complexity in structured documents: User interface issues. In J. J. H. Miller, editor, *PROTEXT IV: Proceedings of the Fourth International Conference on Text Processing Systems*, Boole Press, 1987. To appear.

[Fur88] Richard Furuta. Concepts and models for structured documents. 1988. This book.

[GB79] G. Gaillat and M. Berthod. Panorama des techniques d'extraction de traits caractéristiques en lecture optique de caractères (overview of techniques for extracting characteristic features in optical character recognition). *Revue Thomson-CSF*, 11(4), December 1979.

[Gol78] Charles Goldfarb. *Document Composition Facility: Generalized Markup Language (GML) Users Guide.* IBM General Products Division, 1978. SH20-9160-0.

[Gol81] C. F. Goldfarb. A generalized approach to document markup. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, 16(6):68–73, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981.

[Gol83] Charles F. Goldfarb, editor. *GCA Standard 101–1983, Document Markup Metalanguage: GENCODE (tm) and the Standard Generalized Markup Language (SGML).* Graphic Communications Association, 1983.

[GP82] T. R. G. Green and S. J. Payne. The woolly jumper: typographic problems of concurrency in information display. *Visible Language*, 16, 1982.

[Gri57] H. P. Grice. Meaning. *Philosophical Review*, 6:377–388, 1957.

[Gut85] J. Gutknecht. Concepts of the text editor Lara. *Communications of the ACM*, 28(9):942–960, September 1985.

[GW84] J. Gutknecht and W. Winiger. Andra: The document preparation system of the personal workstation Lilith. *Software—Practice and Experience*, 14:73–100, 1984.

[GW85] H. Ganzinger and W. Wilmertinger. FOAM: A two-level approach to text formatting on a microcomputer system. *Software—Practice and Experience*, 15(4):327–342, April 1985.

[Hal85] M. A. K. Halliday. *An introduction to functional grammar.* Edward Arnold, London, 1985.

[Ham86] Richard Hamlet. A disciplined text environment. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 78–89, Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[Har68] Z. S. Harris. *Mathematical Structures of Language.* John Wiley and Sons, 1968.

[Her84] W. R. Hershey. Thinktank, an outlining and organizing tool. *Byte*, 189–194, May 1984.

[HIAG81a] Michael Hammer, Richard Ilson, Timothy Anderson, Edward J. Gilbert, Michael Good, Bahram Niamir, Larry Rosenstein, and

Sandor Schoichet. The implementation of Etude, an integrated and interactive document production system. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, 16(6):137–141, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981. This report was previously issued as Office Automation Group Memo OAM-026, M.I.T. Laboratory for Computer Science, Cambridge, Mass. (December 1980).

[HIAG81b] Michael Hammer, Richard Ilson, Timothy Anderson, Edward J. Gilbert, Michael Good, Bahram Niamir, Larry Rosenstein, and Sandor Schoichet. *Etude: An integrated document processing system.* Office Automation Group Memo OAM-028, M.I.T. Laboratory for Computer Science, Cambridge, Mass., February 1981. Presented at the 1981 Office Automation Conference, March 23-25, 1981.

[Hib83] P. Hibbard. *User Manual for Mint—The Spice Document Preparation System.* Spice Document S153, Carnegie-Mellon University, April 1983.

[HN82] E. Harslem and L. E. Nelson. A retrospective on the development of Star. In *Proceedings of the 6th International Conference on Software Engineering*, pages 377–383, September 1982. Conference held in Tokyo.

[Hol76] A. W. Holt. The impact of new hardware on OCR design. *Pattern Recognition*, 8:99–105, 1976.

[Hor85a] W. Horak. Office document architecture and office document interchange formats: current status of international standardization. *IEEE Computer*, 18(10):50–62, October 1985.

[Hor85b] Susan B. Horwitz. *Generating Language-Based Editors: A Relationally-Attributed Approach.* Ph.D. dissertation, Cornell University, Department of Computer Science, Ithaca, NY, 1985.

[HT85] Susan Horwitz and Tim Teitelbaum. Relations and attributes: A symbiotic basis for editing environments. *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, SIGPLAN Notices*, 20(7):93–106, July 1985.

[IBM80] IBM. *Document Composition Facility Generalized Markup Language: Concepts and design guide.* IBM Corporation, second

edition, April 1980. Order number SH20-9188-0.

[Ils80] Richard Ilson. *An integrated approach to formatted document production.* Technical Report MIT/LCS/TR-253, M.I.T. Laboratory for Computer Science, Cambridge, Mass., August 1980. M.S. thesis.

[Int84] Interleaf. *Office Publishing System: Introductory User's Manual.* Interleaf, June 1984. Release 1.0, Draft version.

[ISO86a] ISO. *Office Document Architecture.* International Standard Organisation, 1986. Draft International Standard 8813.

[ISO86b] ISO. *Text and Office Systems—Standard Generalized Markup Language.* 1986. Draft International Standard 8879.

[Jol83] V. Joloboff. An interactive graphics editor for document preparation. *ACM SigPC Notices*, 6(6):45–53, 1983. Proceedings of 1983 ACM Conference on Personal and Small Computers, San Diego, December 1983.

[Jol86] Vania Joloboff. Trends and standards in document representation. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 107–124, Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[Jol88] Vania Joloboff. Document representation: Concepts and standards. 1988. This book.

[JQ84] Vania Joloboff and Vincent Quint. Two dimensional editing. In *Proceedings of the IEEE First International Conference on Office Automation*, IEEE Computer Society, 1984. Conference held in New Orleans, Louisiana.

[JSP85] Vania Joloboff, Teo Schleich, and Roy Pierce. *Document Interchange Standard "Interscript".* Document TC97/SC18/N439R, International Standard Organisation, 1985.

[KC75] Brian W. Kernighan and Lorinda L. Cherry. A system for typesetting mathematics. *Communications of the ACM*, 18(3):151–157, March 1975. Also available as Computing Science Technical Report No. 17, Bell Laboratories, Murray Hill, N.J. (Revised, April 1977).

[Kel86] D. Keller-Cohen. Reading periodic bills. *AILA Review*, 3, 1986.

[Ker81a] Brian W. Kernighan. PIC—A language for typesetting graphics. *Proceedings of the ACM SIGPLAN SIGOA Symposium on*

Text Manipulation, SIGPLAN Notices, 16(6):92–98, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981.

[Ker81b] Brian W. Kernighan. *PIC—A crude graphics language for typesetting.* Computing Science Technical Report 85, Bell Laboratories, Murray Hill, N.J., January 1981.

[Ker82] Brian W. Kernighan. PIC—A language for typesetting graphics. *Software—Practice and Experience*, 12(1):1–21, January 1982. A preliminary version of this paper appeared as [Ker81a].

[Ker84] Brian W. Kernighan. *PIC—A Graphics Language for Typesetting, Revised user manual.* Computing Science Technical Report 116, AT&T Bell Laboratories, Murray Hill, N.J., December 1984.

[Kim84] Gary Dean Kimura. *A Structure Editor and Model for Abstract Document Objects.* Ph.D. dissertation, University of Washington, Department of Computer Science, Seattle, WA, 1984. Also available as Technical Report No. 84-07-04, Department of Computer Science, University of Washington (July 1984).

[Kim86] Gary D. Kimura. A structure editor for abstract document objects. *IEEE Transactions on Software Engineering*, SE–12(3):417–435, March 1986.

[Kin84] Peter R. King. W-p: A prototype text-processor and document formatter. In J. J. H. Miller, editor, *PROTEXT I: Proceedings of the First International Conference on Text Processing Systems*, pages 161–172, Boole Press, October 1984.

[Kin86] Peter R. King. An overview of the W document preparation system. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 189–199, Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[KLO78] B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, Jr. UNIX time-sharing system: Document preparation. *The Bell System Technical Journal*, 57(6):2115–2135, July–August 1978.

[KLW66] J. H. Kuney, B. G. Lazorchak, and S. W. Walcavich. Computerized typesetting of complex scientific material. *Proceedings, AFIPS Fall Joint Computer Conference*, 29:149–156, 1966.

[Knu79a] Donald E. Knuth. *TEX and* METAFONT*: New Directions in Type-*

setting. Digital Press and the American Mathematical Society, December 1979.

[Knu79b] Donald E. Knuth. Mathematical typography. In *TEX and* METAFONT*: New Directions in Typesetting*, chapter 1, Digital Press and the American Mathematical Society, December 1979. Reprinted from the *Bulletin* (New Series) *of the American Mathematical Society* 1(2):337-372, March 1979. Josiah Willard Gibbs lectures, presented January 4, 1978.

[Knu84] Donald E. Knuth. *The TEXbook.* Addison-Wesley, 1984.

[Knu85a] Donald E. Knuth. Lessons learned from METAFONT. *Visible Language*, 19(1):35–53, 1985.

[Knu85b] Donald E. Knuth. *TEX: The Program.* Addison-Wesley, 1985.

[KP81] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software—Practice and Experience*, 11(11):1119–1184, November 1981. Also issued as Technical Report No. STAN-CS-80-828, Stanford Department of Computer Science, Stanford, Calif. (November 1980).

[KPB87] S. Kahan, T. Pavlidis, and H. S. Baird. On the recognition of printed characters of any font and size. March 1987.

[KS84] Gary D. Kimura and Alan C. Shaw. The structure of abstract document objects. *Proceedings of the Second ACM-SIGOA Conference on Office Information Systems, SIGOA Newsletter*, 5(1–2):161–169, June 1984. Also issued as Technical Report No. 83-09-02, Department of Computer Science, University of Washington, Seattle, WA, September 1983.

[Lam78] Butler W. Lampson. Bravo manual. In B. W. Lampson and E. A. Taft, editors, *Alto User's Handbook*, Computer Science Laboratory, Xerox Palo Alto Research Center, November 1978.

[Lam85] Leslie Lamport. LATEX*: A Document Preparation System.* Addison-Wesley, 1985.

[LB85] S. Lam and H. S. Baird. *Performance Testing of Mixed-font, Variable-size Character Recognizers.* Technical Report, AT&T Bell Labs, 1985.

[LCMO81] Butler W. Lampson, Douglas W. Clark, Gene A. McDaniel, Severo M. Ornstein, and Kenneth A. Pier. *The Dorado: A High-Performance Personal Computer—Three Papers.* Technical Report CSL-81-1, Xerox Palo Alto Research Center, Palo

Alto, Calif., January 1981. Includes [LP80, LMO81, CLP81].

[Les76] M. E. Lesk. *Tbl—A program to format tables.* Computing Science Technical Report 49, Bell Laboratories, Murray Hill, N.J., September 1976.

[Les78] M. E. Lesk. *Some applications of inverted indexes on the UNIX System.* Computing Science Technical Report 69, Bell Laboratories, Murray Hill, N.J., June 1978.

[Lev83] Michael Levison. Editing mathematical formulæ. *Software—Practice and Experience*, 13:189–195, 1983.

[LG16] L. A. Legros and J. C. Grant. *Typographical printing-surfaces.* Longmans, Green, London, 1916.

[LMO81] Butler W. Lampson, Gene A. McDaniel, and Severo M. Ornstein. An instruction fetch unit for a high-performance personal computer. January 1981. Contained in [LCMO81].

[LP80] Butler W. Lampson and Kenneth A. Pier. A processor for a high-performance personal computer. *Proceedings of the 7th Symposium on Computer Architecture*, May 1980. Also contained in [LCMO81].

[Mac81] Ian A. Macleod. A data base management system for document retrieval applications. *Information Systems*, 6(2):131–137, 1981.

[Man86] J. Mantas. An overview of character recognition methodologies. *Pattern Recognition*, 19(6):425–430, 1986.

[Mat77] Mathlab Group. *MACSYMA Reference Manual.* Laboratory for Computer Science, M.I.T., December 1977. Version Nine, Second Printing.

[Mat78] Mathlab Group. *MACSYMA Primer.* Laboratory for Computer Science, M.I.T., March 1978.

[Med82] Raul Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments.* Ph.D. dissertation, Carnegie-Mellon University Computer Science Department, Pittsburgh, PA, March 1982. Also issued as Technical Report CMU-CS-82-113.

[Mel86] B. Mélèse. Edition de documents multi-langages sous Mentor-Rapport. *T.S.I.*, 4(5):267–277, 1986.

[Mel87] B. Mélèse. Editing multi-langage documents under Mentor-Rapport. *T.S.I. (English edition)*, 6(1):33–42, 1987. John Wiley, publisher. English translation of [Mel86].

[Men84] Jerry Mendelson. *Interpress 82 Reader's Guide.* Xerox System Integration Guide XSIG 018404, Xerox Corporation, El Segundo, Calif., April 1984.

[MF81] Raul Medina-Mora and Peter H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, SE-7(5):472–482, September 1981.

[MN81] Raúl Medina-Mora and David S. Notkin. *ALOE Users' and Implementors' Guide.* Technical Report CMU-CS-81-145, Department of Computer Science, Carnegie-Mellon University, November 1981.

[Moj86] M. Mojahid. *Eléments d'un langage de représentation de connaissances éditoriales pour un système multi-agents de restitution de textes structurés (Elements of a Representation Language for Editorial Knowledge, for a Multi-agent System for the Restitution of Structured Texts).* Research Diploma report, LSI, UPS, June 1986.

[Mor85] Robert A. Morris. Is what you see enough to get? A description of the Interleaf publishing system. In J. J. H. Miller, editor, *PROTEXT II: Proceedings of the Second International Conference on Text Processing Systems*, pages 56–81, Boole Press, October 1985.

[MvD82] Norman Meyrowitz and Andries van Dam. Interactive editing systems: Parts I and II. *ACM Computing Surveys*, 14(144):321–415, September 1982.

[NN84] J. Nanard and M. Nanard. Manipulation interactive de documents. *T.S.I.*, 3(6):443–451, 1984.

[NNC86] J. Nanard, M. Nanard, and G. Cottin. Pléiade, a system for interactive manipulation of structured documents. In *Conference on Workstations and Publication Systems*, British Computer Society, October 1986. Conference held in London, October 22-23, 1986.

[Nor84] P. Norrish. Moving tables from paper to screen. *Visible Language*, 18, 1984.

[Nor87] P. Norrish. *The graphic translatability of text.* British Library R & D report no. 5854, British Library, London, London, 1987. Department of Typography & Graphic Communication, University of Reading, England.

[Not84] David Notkin. *Interactive Structure-Oriented Computing.* Ph.D. dissertation, Carnegie-Mellon University Computer Science Department, Pittsburgh, PA, February 1984. Also issued as Technical Report CMU-CS-84-103.

[Not85a] David Notkin. The GANDALF project. *The Journal of Systems and Software*, 5(2):91–105, May 1985.

[Not85b] David Notkin. Annotated bibliography of GANDALF literature. *The Journal of Systems and Software*, 5(2):173–176, May 1985.

[NSS85] G. Nagy, S. C. Seth, and S. D. Scoddard. Document analysis with an expert system. In *Pattern recognition in practice, Amsterdam*, pages 149–159, 1985.

[Pal86] R. S. Palais. Mathematical text processing. *Notices of the AMS*, 3–37, January 1986.

[Pie72] John Pierson. *Computer Composition using PAGE-1.* Wiley-Interscience, 1972.

[Pie83] Kenneth A. Pier. *A Retrospective on the Dorado, A High-Performance Personal Computer.* Technical Report ISL-83-1, Xerox Palo Alto Research Center, Palo Alto, Calif., August 1983.

[PJN85] Arno J. H. M. Peels, Norbert J. M. Janssen, and Wop Nawijn. Document architecture and text formatting. *ACM Transactions on Office Information Systems*, 3(4):347–369, October 1985.

[PK82] Michael F. Plass and Donald E. Knuth. Choosing better line breaks. In Jurg Nievergelt, Giovanni Coray, Jean-Daniel Nicoud, and Alan C. Shaw, editors, *Document Preparation Systems: A Collection of Survey Articles*, pages 221–242, North-Holland Publishing Company, 1982.

[Pla81] Michael F. Plass. *Optimal Pagination Techniques for Automatic Typesetting Systems.* Ph.D. dissertation, Stanford University Computer Science Department, Stanford, CA, June 1981. Also issued as Stanford Technical Report No. STAN-CS-81-870 (June 1981) and Xerox Palo Alto Research Center Technical Report ISL-81-1 (August 1981).

[Qui83a] V. Quint. An interactive system for mathematical text processing. *T.S.I. (English edition)*, 2(3):169–179, 1983. NOA, publisher. English translation of [Qui83b].

[Qui83b] V. Quint. Un système interactif pour la production des docu-

ments mathématiques. *T.S.I.*, 2(3):179–190, 1983.

[Qui84] Vincent Quint. Interactive editing of mathematics. In J. J. H. Miller, editor, *PROTEXT I: Proceedings of the First International Conference on Text Processing Systems*, pages 55–68, Boole Press, October 1984.

[Qui88] Vincent Quint. Systems for the manipulation of structured documents. 1988. This book.

[QV86] Vincent Quint and Irène Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 200–213, Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[QVB86a] V. Quint, I. Vatton, and H. Bedor. Le système grif. *T.S.I.*, 5(4):337–341, 1986.

[QVB86b] Vincent Quint, Irène Vatton, and Hassan Bedor. Grif: An interactive environment for TEX. In Jacques Désarménien, editor, *TEX for Scientific Documentation*, pages 145–158, Springer-Verlag, 1986. Lecture notes in Computer Science, No. 236.

[QVB87] V. Quint, I. Vatton, and H. Bedor. The Grif system. *T.S.I. (English edition)*, 6(1):98–103, 1987. John Wiley, publisher. English translation of [QVB86a].

[Rei80a] Brian K. Reid. *Scribe: A Document Specification Language and its Compiler.* Ph.D. dissertation, Carnegie-Mellon University Computer Science Department, Pittsburgh, PA, October 1980. Also issued as Technical Report CMU-CS-81-100.

[Rei80b] Brian K. Reid. A high-level approach to computer document formatting. *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, January 1980.

[Rei81a] B. K. Reid. The Scribe document specification language and its compiler. *Abstracts of the Presented Papers, International Conference on Research and Trends in Document Preparation Systems,* Lausanne, Switzerland, 59–62, February 1981.

[Rei81b] Phyllis Reisner. Formal grammar and human factors design of an interactive graphics system. *IEEE Transactions on Software Engineering*, SE–7(2):229–240, March 1981.

[Rei83] Brian Reid. Scribe: Histoire et évaluation. In Jacques André, editor, *Actes des Journées sur la Manipulation de Documents*, pages 28–39, INRIA/IRISA, May 1983.

[Rei84] Steven P. Reiss. Graphical program development with PECAN. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Software Engineering Notes*, 9(3):30–41, May 1984. Also *SIGPLAN Notices* 19(3), May 1984.

[Rei86a] Brian Reid. PostScript and Interpress. In *Documentation Graphics, SIGGRAPH '86 Tutorial*, 1986. Presented at ACM SIGGRAPH 1986.

[Rei86b] Brian K. Reid. Procedural page description languages. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 214–223, Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[Rit78] D. M. Ritchie. UNIX time sharing system: A retrospective. *The Bell System Technical Journal*, 57(6):1947–1969, July–August 1978.

[RS86] A. Rastogi and S. N. Srihari. *Recognizing textual blocks using the Hough transform.* Technical Report 86-01, Department of Computer Science, SUNY/Buffalo, 1986.

[RW80] Brian K. Reid and Janet H. Walker. *Scribe Introductory User's Manual*, Third Edition, Preliminary Draft. UniLogic, Ltd., Pittsburgh, May 1980. Previous editions were issued by the Carnegie-Mellon University Computer Science Department, Pittsburgh, PA.

[Sal65] J. Saltzer. Manuscript typing and editing: TYPSET, RUNOFF. In P. A. Crisman, editor, *The Compatible Time-Sharing System: A programmer's guide*, Second Edition, chapter AH.9.01, The MIT Press, 1965.

[SBB80] R. J. Spiro, B. Bruce, and W. Brewer, editors. *Theoretical Issues in Reading Comprehension, Perspective from Cognitive Psychology, Linguistics, Artificial Intelligence and Education.* Lawrence Erlbaum, 1980.

[SBM80] C. Y. Suen, M. Berthod, and S. Mori. Automatic recognition of handprinted characters—The state of the art. *IEEE Proceedings*, (68), April 1980.

[Sch81] S. Schoichet. Page layout: Specification and representation. *Abstracts of the Presented Papers, International Conference on Research and Trends in Document Preparation Systems,* Lausanne, Switzerland, 23–28, February 1981.

[Sco85] Jeffrey Alan Scofield. *Editing as a Paradigm for User Interaction.* Ph.D. dissertation, University of Washington, Department of Computer Science, Seattle, WA, 1985. Also available as Technical Report No. 85-08-10, Department of Computer Science, University of Washington (August 1985).

[Sea75] J. R. Searle. A taxonomy of illocutory speech acts. In K. Gunderson, editor, *Language, Mind and Knowledge*, pages 334–336, University of Minesota Press, 1975.

[Sea80] L. Searle. *Speech Act Theory and Pragmatics.* Reidel, 1980.

[Sey81] Jonathan Seybold. Xerox's 'Star'. *The Seybold Report*, 10(16), April 27, 1981.

[SFS80] Alan Shaw, Richard Furuta, and Jeffrey Scofield. *Document Formatting Systems: Survey, Concepts, and Issues [Extended Abstract].* Technical Report 80-10-02, University of Washington, Department of Computer Science, Seattle, WA, October 1980. Also available in the *Abstracts of the Presented Papers, International Conference on Research and Trends in Document Preparation Systems*, Lausanne, Switzerland, 47-52, February 1981.

[Sha78] Alan C. Shaw. Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, SE-4(3):242–254, May 1978.

[Sha79] Alan C. Shaw. *Software Specification Languages Based on Regular Expressions.* Berichte des Instituts für Informatik 31, Eidgenössische Technische Hochschule Zürich, Zürich, Switzerland, June 1979.

[Sha80] Alan C. Shaw. *A Model for Document Preparation Systems.* Technical Report 80-04-02, University of Washington, Department of Computer Science, Seattle, WA, April 1980.

[SIKV82] David Canfield Smith, Charles Irby, Ralph Kimball, and Bill Verplank. Designing the Star user interface. *Byte*, 7(4):242–282, April 1982.

[Sop86] C. Sopena. *Editions des structures textuelles d'un document*

intelligemment dirigée par un module de contrôle (Editing of Document Textual Structures Intelligently Guided by a Control Module). Research diploma report, LSI, UPS, June 1986.

[Sou84] R. Southall. First principles of typographic design for document production. *TUGboat*, 5:79–90, 1984.

[Sou85] R. Southall. *Designing new typefaces with* METAFONT. Report Number STAN-CS-85-1074, Stanford Department of Computer Science, Stanford, Calif., 1985.

[Sou86] Richard Southall. Designing a new typeface with METAFONT. In Jacques Désarménien, editor, *TEX for Scientific Documentation*, pages 161–179, Springer-Verlag, 1986. Lecture notes in Computer Science, No. 236.

[Spi80] Michael Spivak. *The Joy of TEX: A gourmet guide to typesetting technical text by computer,* Version -1. American Mathematical Society, 1980.

[SR84] Robert F. Sproull and Brian K. Reid. *Introduction to Interpress.* Xerox System Integration Guide XSIG 038404, Xerox Corporation, El Segundo, Calif., April 1984.

[Sta81] Richard M. Stallman. EMACS, the extensible, customizable self-documenting display editor. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, 16(6):147–156, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981. This report is a revised version of AI Memo Number 519, M.I.T. Artificial Intelligence Lab., Cambridge, Mass. (June 1979).

[STI83] STI. *Reference Manual for Copy Editors, Production Assistants, and Keyboarders in Computer-Assisted Photocomposition.* Science Typographers, Inc., 1983. Sixth Edition.

[Str71] P. F. Strawson. *Logico-linguistic papers.* Methuen, London, 1971.

[SZ86] S. N. Srihari and G. W. Zack. Document image analysis. In *Proceedings of the 8th International Conference on Pattern Recognition, Paris*, pages 434–436, 1986.

[Taz85] S. Tazi. *Représentation de connaissances structurelles du texte naturel pour la conception d'un éditeur "intelligent" (Representation of Structural Knowledge of Natural Texts for the Design*

of an Intelligent Editor). Ph.D. dissertation, UPS, 1985.

[Tei84a] Warren Teitelman. A tour through Cedar. *IEEE Software*, 1(2):44–73, April 1984.

[Tei84b] Warren Teitelman. *The Cedar Programming Environment: A Midterm Report and Examination.* Technical Report CSL-83-11, Xerox Palo Alto Research Center, Palo Alto, Calif., July 1984.

[Tei85] Warren Teitelman. A tour through Cedar. *IEEE Transactions on Software Engineering*, SE-11(3):285–302, March 1985.

[Tes72] Larry Tesler. *PUB: The document compiler.* Operating Note 70, Stanford Artificial Intelligence Project, Stanford, Calif., September 1972.

[TMLS79] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. *Alto: A personal computer.* Technical Report CSL-79-11, Xerox Palo Alto Research Center, Palo Alto, Calif., August 1979. Also contained in Siewiorek, Bell, and Newell, *Computer Structures: Readings and Examples*, second edition.

[TR81] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 563–573, September 1981.

[TV85a] S. Tazi and J. Virbel. *Is the hierarchical model powerful enough for textual structures representation?* LSI, Toulouse, 1985.

[TV85b] S. Tazi and J. Virbel. Formal representation of textual structures for an intelligent text-editing system. In V. Dahl and P. Saint-Dizier, editors, *Natural Language Understanding and Logic Programming*, pages 191–205, North-Holland, 1985.

[Uni84] Unilogic, Ltd. *Scribe Introductory User's Manual.* Unilogic, Ltd., Pittsburgh, fourth edition, April 1984.

[Utt85] Ian Utting. The rôle of device independent output. In Dario Lucarella, editor, *Proceedings of the First European Conference on TEX for Scientific Documentation*, pages 183–189, Addison-Wesley, 1985.

[Vir82] J. Virbel. La composante matérielle des structures textuelles (the concrete component of textual structures). In A. Borillo, M. Borillo, L. Farinas del Cerro, and J. Virbel, editors, *Approches formelles de la sémantique naturelle (Formal Approaches to Nat-*

ural Semantics), LSI, Toulouse, 1982.

[Vir85] J. Virbel. Représentation et utilisation de connaissances textuelles (representation and use of textual knowledge). In *Cognitiva, Paris*, pages 879–883, June 1985.

[Vir86] J. Virbel. *Langage et métalangage dans le texte du point de vue de l'édition en informatique textuelle (Language and metalanguage in texts from the point of view of editing within textual computer science).* Cahiers de Grammaire 10, Université Toulouse-Le Mirail, Toulouse, France, 1986.

[Vir88] Jacques Virbel. The contribution of linguistic knowledge to the interpretation of text structures. 1988. This book.

[VV73] J. C. Verges and J. P. Verjus. Reconnaissance automatique des structures des textes en vue de l'édition (automatic recognition of text structures for editing). *R. A. I. R. O.*, B–2:85–120, October 1973.

[vV86] J. C. van Vliet, editor. *Text Processing and Document Manipulation.* Cambridge University Press, April 1986. Proceedings of the international conference, University of Nottingham, 14–16 April 1986.

[vV88] Hans van Vliet, editor. *Document Manipulation and Typography.* Cambridge University Press, Oxford, England, April 1988. Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography, Nice (France), April 20-22, 1988.

[VW80] Christopher John Van Wyk. *A Language for Typesetting Graphics.* Ph.D. dissertation, Stanford University, Stanford, Calif., June 1980.

[VW81a] Christopher J. Van Wyk. A graphics typesetting language. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, 16(6):99–107, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981.

[VW81b] Christopher J. Van Wyk. *IDEAL User's Manual.* Computing Science Technical Report 103, Bell Laboratories, Murray Hill, N.J., December 1981.

[VW82] Christopher J. Van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April

1982.

[Wal80] R. W. Waller. Graphic aspects of complex texts: Typography as macro-punctuation. In P. A. Kolers, M. E. Wrolstad, and H. Bouma, editors, *Processing of visible language 2*, Plenum, New York, 1980.

[Wal81] Janet H. Walker. The document editor: A support environment for preparing technical documents. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices*, 16(6):44–50, June 1981. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2(1&2), Spring/Summer 1981.

[Wal83] S. Walker. *Descriptive techniques for studying verbal graphic language.* Ph.D. thesis, Department of Typography & Graphic Communication, University of Reading, 1983.

[Whi83] Sam Whidden. Observations on TEX from a divergent viewpoint: A critical commentary. *TUGboat*, 4(2):81–102, 1983. Comparisons between STI (see [STI83]) and TEX (see [Knu84]) with responses by J. R. Roesser, Don Knuth, David Fuchs, Michael Spivak, Richard Palais, and Barbara Beeton.

[Wil84] G. Williams. The Apple Macintosh computer. *Byte*, 30–54, February 1984.

[WT80] P. Wright and M. S. Threfell. Readers' expectations about format influence the usability of an index. *Journal of Research Communication Studies*, 2, 1980.

[WW82] John Warnock and Douglas K. Wyatt. A device independent graphics imaging model for use with raster devices. *SIGGRAPH '82 Conference Proceedings, Computer Graphics*, 16(3):313–319, July 1982.

[Xer84a] Xerox. *Interpress Electronic Printing Standard.* Xerox Corporation, Stamford Connecticut, April 1984. Xerox System Integration Standard, XSIG 048404, Version 2.1.

[Xer84b] Xerox. *8010 STAR Information System Reference Library.* Xerox Corporation, 1984.

[Xer85] Xerox. *Interpress Electronic Printing Standard.* Xerox Corporation, Stamford Connecticut, December 1985. Xerox System Integration Standard, XSIG 048512, Version 3.0.